Introduction & Procedural Programming

Sample Courseware

Introduction to Software Paradigms & Procedural Programming Paradigm

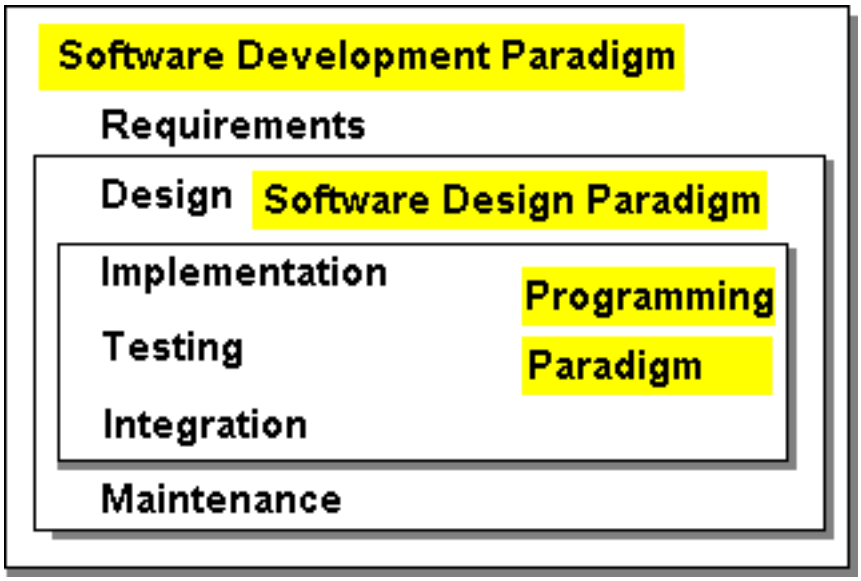
This Lesson introduces main terminology to be used in the whole course. Thus, we discuss the terms "Programming Language", "Software Paradigm", "Software Architecture", "Design Pattern" and "Software Engineering" from a general perspective.
The lesson breafly discusses a well-known procedural programming paradigm and main principles of software engineering.

1 Introduction

"**Paradigm**" (a Greek word meaning example) is commonly used to refer to a category of entities that share a common characteristic.

Thus, we can distinguish between: Programming Paradigms, Software Design Paradigms and Software Development Paradigms.

- A **Programming Paradigm** is a model for a class of Programming Languages that share a set of common characteristics.
- **Software Design Paradigm** embody the results of people's ideas on how to construct programs, combine them into large software systems and formal mechanisms for how those ideas should be expressed.
- **Software Development Paradigm** is also known as Software Engineering, defines common principles of managing big software projects, especially when a big number of employees with different area of expertize is engaged.



1.1 Programming Paradigms

Definition 1:

> A **Programming Paradigm** is a model for a class of Programming Languages that share a set of common characteristics.

Definition 2:

> The programming paradigm is a general principles that are used by a programmer to communicate a task/algorithm to a computer.

Definition 3:

> The programming paradigm is a general way of thinking when a particular computation is implemented.

Question: *Why do we need to know programming paradigms?*

Answer 1: ***Quite simple, we cannot understand a programming language without knowing a programming paradigm !***

A programming language is a system of signs used to communicate a task/algorithm to a computer, causing the task to be performed. The task to be performed is called a computation, which follows absolutely precise and unambiguous rules.

There are three crucial components to any language.

- The **language paradigm** is a general principles that are used by a programmer to communicate a task/algorithm to a computer.
- The **syntax** of the language is a way of specifying what is legal in the phrase structure of the language; knowing the syntax is analogous to knowing how to spell and form sentences in a natural language like English. However, this doesn't tell us anything about what the sentences mean.
- The third component is **semantics**, or meaning, of a program in that language. Ultimately, without a semantics, a programming language is just a collection of meaningless phrases.

Question: *Why do we need to know programming paradigms?*

Answer 2: ***Generally, a selected Programming Paradigm defines main property of a software developed by means of a programming language supporting the paradigm.***
Hence, selecting of a proper programming paradigm is essential.

We always take into consideration the following properties of a software system:

- scalability/modifiability
- integrability/reusability
- portability
- performance
- reliability
- ease of creation

Question: *Why do we need to know programming paradigms?*

Answer 3: ***Programming Paradigm is a way of programmers' thinking. Thus, a programming paradigm influences all steps of software development and software***

design paradigms *(see below)*

For example, the phases in a software lifecycle:

- Specification (analysis)
- Design
- Implementation
- Testing
- Integration
- Maintenance

may be seen differently in context of different programming paradigms.

Question: *There have been a large number of programming languages. Back in the 60's there were over 700 of them. Do we need to overview 700 programming paradigms?*

Answer: ***Fortunately, there are just four major programming language paradigms:***

- **Imperative (Procedural)** Paradigm (Fortran, C, Ada, etc.)
- **Object-Oriented** Paradigm (SmallTalk, Java, C++)
- **Functional** Paradigm (Lisp, ML, Haskell)
- **Logic** Paradigm (Prolog)

1.2 Software Design Paradigms

Software Design Paradigm embody the results of people's ideas on how to construct programs, combine them into large software systems and formal mechanisms for how those ideas should be expressed.
Thus, we can say that a Software Design Paradigm is a model for a class of problems that share a set of common characteristics.

Software design paradigms can be sub-divided as:

- **Design Patterns**
- **Components**

- **Software Architecture**
- **Frameworks**

It should be especially noted that a particular Programming Paradigm essentially defines software design paradigms. For example, we can speak about Object-Oriented design patterns, procedural components (modules), functional software architecture, etc.

A **design pattern** is a proven solution for a general design problem. It consists of communicating 'objects' that are customized to solve the problem in a particular context. Software design patterns can be sub-divided as:

Architectural Pattern expresses a fundamental structural organization or schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Note, architectural patterns often equated to software architecture

Design Pattern provides a schema for refining the subsystems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

Note, design patterns often equated to software components

Idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components using the features of the given language.

Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning program.

Note, software components often equated to design patterns with Emphasis on reusability.

A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces...typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations

A component must be compatible and interoperate with a whole range of other components.

Examples of components: "Window", "Push Button", "Text Editor", etc.
Two main issues arise with respect to interoperability information:

- how to express interoperability information (e.g. how to add a "push button" to a "window";
- how to publish this information (e.g. library with API reusable via an "include" statement)

> **Software architecture** is the structure of the components of the solution. A particular software architecture decomposes a problem into smaller pieces and attempts to find a solution (Component) for each piece.
> Note, software architecture often equated to architecture patterns.

We can also say that an architecture defines a software system components, their integration and interoperability:

- Integration means the pieces fit together well.
- Interoperation means that they work together effectively to produce an answer.

There are many software architectures. Choosing the right one can be a difficult problem in itself.

> **A software framework** is a reusable mini-architecture that provides the generic structure and behavior for a family of software abstractions, along with a context of metaphors which specifies their collaboration and use within a given domain.
> Note, many authors do not distinguish between software architecture and software framework.

Frameworks can be seen as an intermediate level between components and a software architecture.
Example: Suppose an architecture of a WBT system reuse such components as "Text Editing Input object" and "Push buttons". A software framework may define an "HTML Editor" which can be firther reused for building the system.

2 Software Engineering

> The term "software engineering" was coined in about 1969 to mean "the establishment and use of sound engineering principles in order to economically

obtain software that is reliable and works efficiently on real machines".

This view opposed uniqueness and "magic" of programming in an effort to move the development of software from "magic" (which only a select few can do) to "art" (which the talented can do) to "science" (which supposedly anyone can do!). There have been numerous definitions given for software engineering (including that above and below).

Software Engineering is not a discipline; it is an aspiration, as yet unachieved. Many approaches have been proposed including reusable components, formal methods, structured methods and architectural studies. These approaches chiefly emphasize the engineering product; the solution rather than the problem it solves.

2.1 Software Development Cycle

Current situation with Software Development:

- People developing systems were consistently wrong in their estimates of time, effort, and costs

- Reliability and maintainability were difficult to achieve

- Delivered systems frequently did not work

```
1979 study of a small number of government projects showed that:
  2% worked
  3% could work after some corrections
  45% delivered but never successfully used
  20% used but extensively reworked or abandoned
  30% paid and undelivered
```

- Fixing bugs in delivered software produced more bugs

- Increase in size of software systems

```
  NASA
  StarWars Defense Initiative
```

Social Security Administration
financial transaction systems

- Changes in the ratio of hardware to software costs

early 60's - 80% hardware costs
middle 60's - 40-50% software costs
today - less than 20% hardware costs

- Increasingly important role of maintenance

Fixing errors, modification, adding options
Cost is often twice that of developing the software

- Advances in software techniques (e.g., users interaction)

- Increased demands for software

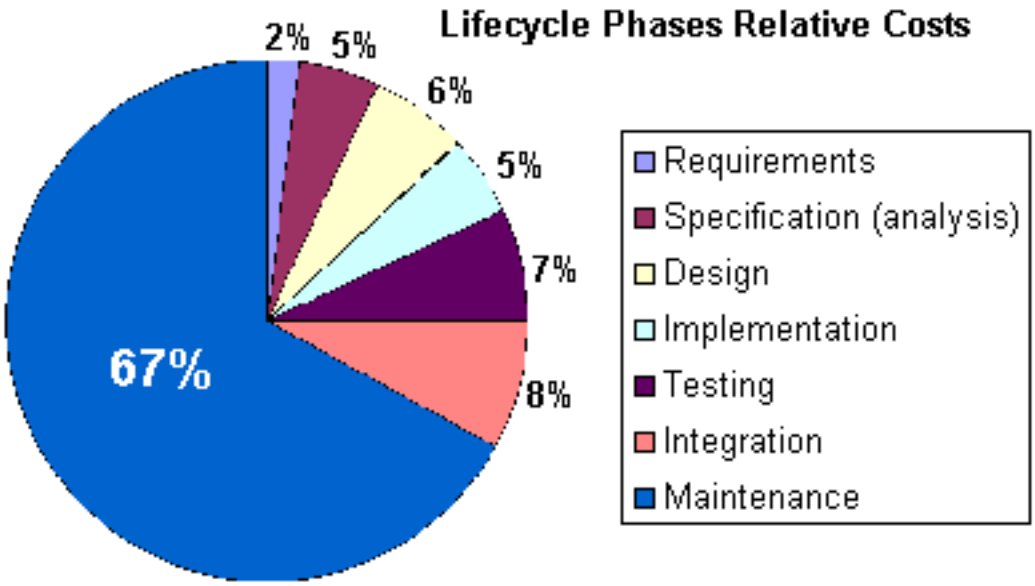
Medicine, Manufacturing, Entertainment, Publishing

- Demand for larger and more complex software systems

Airplanes (crashes), NASA (aborted space shuttle launches),
"ghost" trains, runaway missiles,
ATM machines (have you had your card "swallowed"?), life-support systems, car systems, etc.
US National security and day-to-day operations are highly dependent on computerized systems.

Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a software lifecycle.
Steps or phases in a software lifecycle fall generally into these categories:

- Requirements (Relative Cost 2%)
- Specification (analysis) (Relative Cost 5%)
- Design (Relative Cost 6%)
- Implementation (Relative Cost 5%)
- Testing (Relative Cost 7%)
- Integration (Relative Cost 8%)
- Maintenance (Relative Cost 67%)
- Retirement



Software engineering employs a variety of paradigms, tools, and methods.

> Paradigms refer to particular approaches or philosophies for designing, building and maintaining software.
> Different paradigms each have their own advantages and disadvantages which make one more appropriate in a given situation than perhaps another (!).

A method (also referred to as a technique) is heavily depended on a selected paradigm and may be seen as a paradigm implementation for producing some result. Methods generally involve some formal notation and process(es).
Tools are automated systems implementing a particular method.
Thus, the following phases are heavily affected by selected programming and software design paradigms:

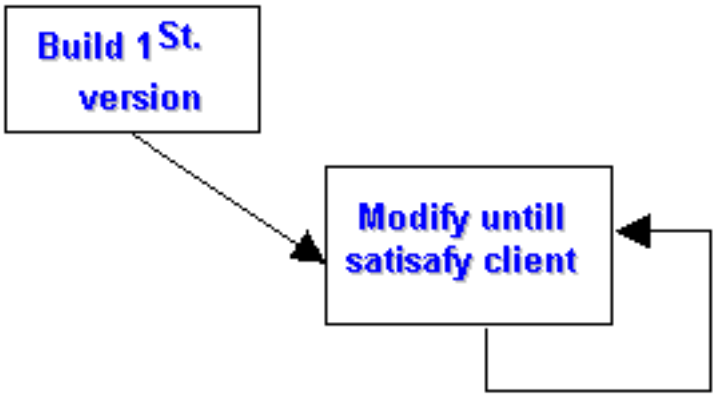
- Design
- Implementation
- Integration
- Maintenance

2.2 Software Development Cycle

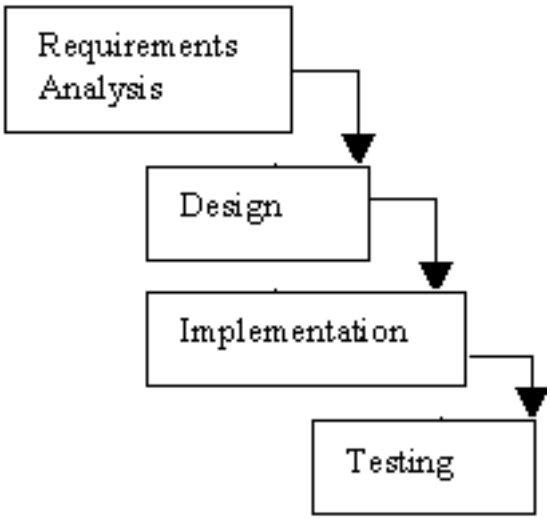
The software development cycle involves the activities in the production of a software system.
Generally the software development cycle can be divided into the following phases:

- **Requirements analysis and specification**
- **Design**
  - Preliminary design
  - Detailed design
- **Implementation**
  - Component Implementation
  - Component Integration
  - System Documenting
- **Testing**
  - Unit testing
  - Integration testing
  - System testing
  - Installation and Acceptance Testing
- **Maintenance**
  - Bug Reporting and Fixing
  - Change requirements and software upgrading

Software development models that will be briefly reviewed include:



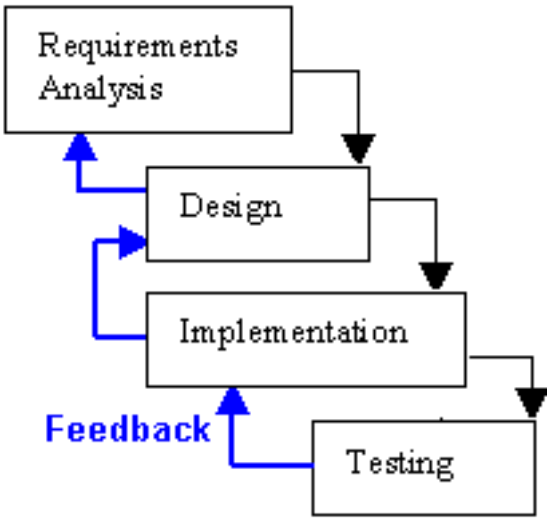
Build and Fix Model:
This works OK for small, simple systems, but is completely unsatisfactory for software systems of any size.
It has been shown empirically that the cost of changing a software product is relatively small if the change is made at the requirements or design phases but grows large at later phases.
The cost of this process model is actually far greater than the cost of a properly specified and designed project. Maintenance can also be problematic in a software system developed under this scenario.



Waterfall Model:

Derived from other engineering processes in 1970. Offered a means of making the development process more structured. Expresses the interaction between subsequent phases.
Each phase cascades into the next phase. In the original waterfall model, a strict sequentially was at least implied. This meant that one phase had to be completed before the next phase was begun.

It also did not provide for feedback between phases or for updating/re-definition of earlier phases.
Implies that there are definite breaks between phases, i.e., that each phase has a strict, non- overlapping start and finish and is carried out sequentially.
Critical point is that no phase is complete until the documentation and/or other products associated with that phase are completed.

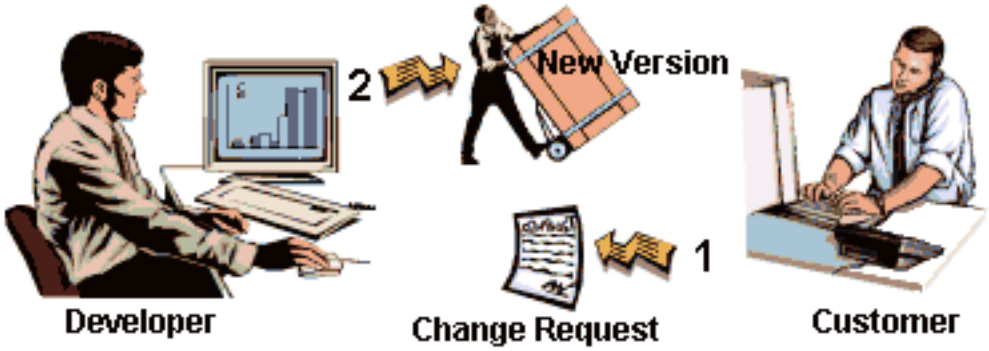


Modified Waterfall Model:

Needed to provide for overlap and feedback between phases. Rather than being a simple linear model, it needed to be an iterative model. To facilitate the completion of the goals, milestones, and tasks, it is normal to freeze parts of the development after a certain point in the iteration.

Verification and validation are added. Verification checks that the system is correct (building the system right). Validation checks that the system meets the users desires (building the right system).

Rapid Prototyping Model:



Prototyping also referred to as evolutionary development, prototyping aims to enhance the accuracy of the designer's perception of the user's requirements. Prototyping is based on the idea of developing an initial implementation for user feedback, and then refining this prototype through many versions until an satisfactory system emerges.

Prototyping allows the clarification of users requirements through, particularly, the early development of the user interface. The user can then try out the system, albeit a (sub) system of what will be the final product. This allows the user to provide feedback before a large investment has been made in the development of the wrong system.
There are two types of prototypes:

- **Exploratory programming**: Objective is to work with the user to explore their requirements and deliver a final system. Starts with the parts of the system which are understood, and then evolves as the user proposes new features.
- **Throw-away prototyping**: Objective is to understand the users' requirements and develop a better requirements definition for the system. Concentrates on poorly understood components.

Boehm’s Spiral Model:

Need an improved software lifecycle model which can subsume all the generic models discussed so far. Must also satisfy the requirements of management.

Boehm proposed a spiral model where each round of the spiral

- identifies the sub problem which has the highest risk associated with it
- finds a solution for that problem.

- Waterfall - Document Driven
- Prototyping - Implementation Driven
- Spiral - Project Mangement Driven

3 Procedural Programming Paradigm

Any imperative program consists of

- **Declarative statements** which gives a name to a value.
- **Imperative statements** which assign new values to variables
- Program **control flow statements** which define order in which imperative statements are evaluated.

Example:

```
var factorial = 1;     /*Declarative statement*/
var argument = 5;
var counter = 1;
while (counter <= argument)   /* flow control statement*/
   {
   factorial = factorial*counter;     /*Imperative statement*/
   counter++;
   }
```

3.1 Variables and Data Types

Different variables in a program may have different types. For example, a language may treat a two bytes as a string of characters and as a number. Dividing a string "20" by number 2 may not be possible.

A language like this has at least two types - one for strings and one for numbers.

```
Example:
var PersonName = new String(); /*variable type "string"*/
var PersonSalary = new Integer(); /*variable type "integer"*/
```

Types can be **weak** or **strong**.

- Strong type means that at any point in the program, when it is running, the type of a

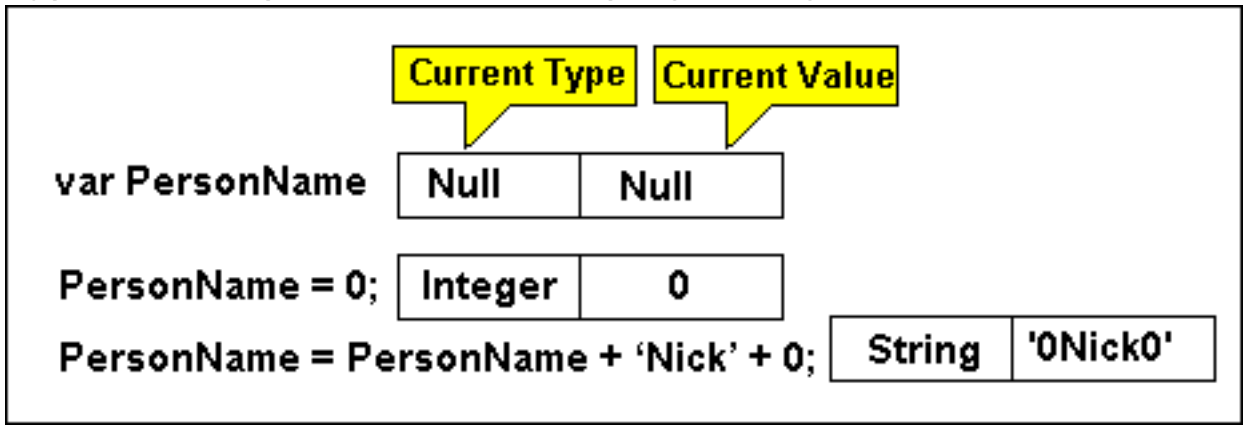
particular chunk of data (i.e. variable) is known.
- Weak type means that imperative operators may change a variable type.

```
Example:
var PersonName; /*variable of a weak type */
PersonName = 0; /*PersonName is an "integer"*/
PersonName = 'Nick'; /*PersonName is a "string"*/
```

Obviously, languages supporting weak variable types need sophisticated rules for type conversions.

```
Example:
var PersonName; /*variable of a weak type*/
PersonName = 0; /*PersonName is an "integer"*/
PersonName = PersonName + 'Nick' + 0; /*PersonName is a string "0Nick0"*/
```

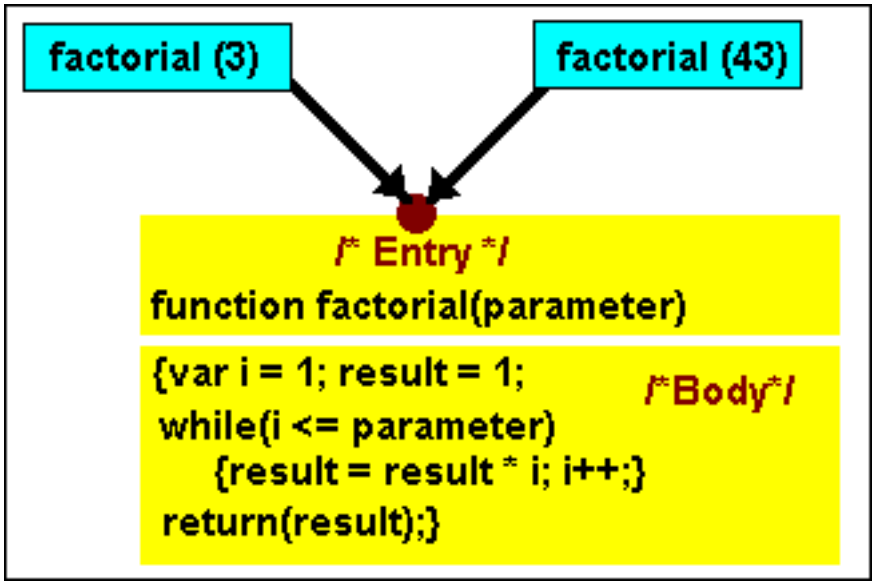
To support weak typing, values are boxed together with information about their type - value and type are then passed around the program together.



3.2 Procedures (Functions)

Programmers have dreamed/attempted of building systems from a library of reusable software components bound together with a little new code.

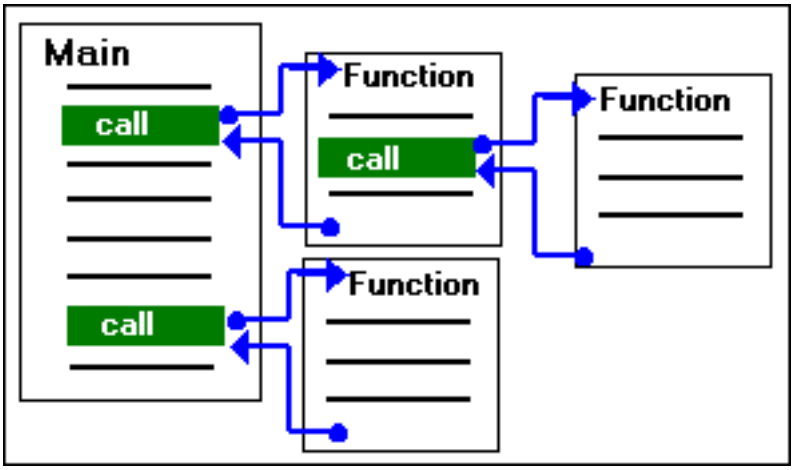
Imperative (Procedural) Programming Paradigm is essentially based on concept of so-called "Modules" also known as "Functions", "Procedures" or "Subroutines".
A module is a section of code that is parceled off from the main program and hidden behind an interface.



The code within the module performs a particular activity, here generating a factorial value. The idea of parceling the code off into a subroutine is to provide a single point of entry.
Anyone wanting a new factorial value has only to call the "factorial" function with the appropriate parameters.

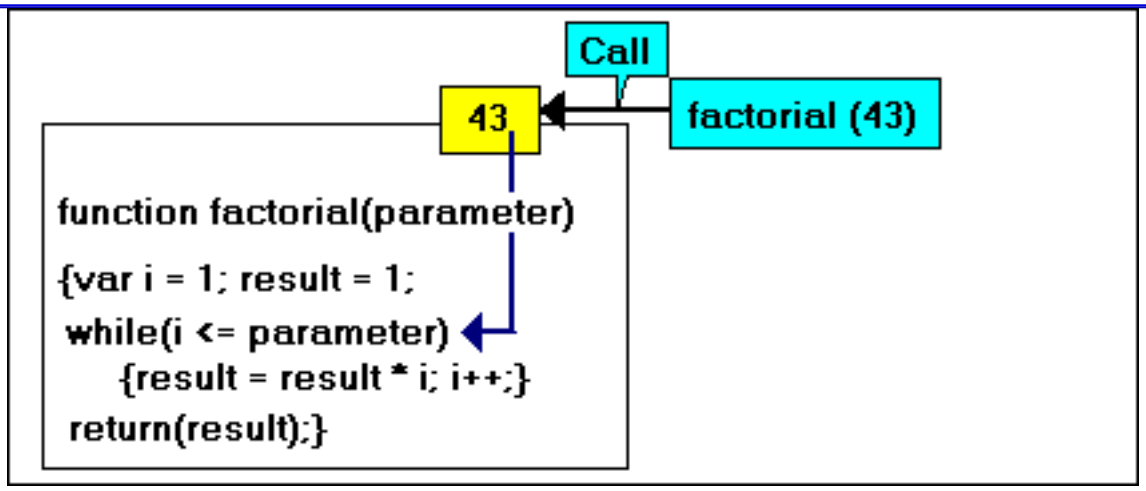
Here's what the conventional application based on the Imperative (Procedural) Programming Paradigm looks like:

- Main procedure determines the control flow for the application
- Functions are called to perform certain tasks or specific logic
- The main and sub procedures that comprise the implementation are structured as a hierarchy of tasks.
- The source for the implementation is compiled and linked with any additional executable modules to produce the application



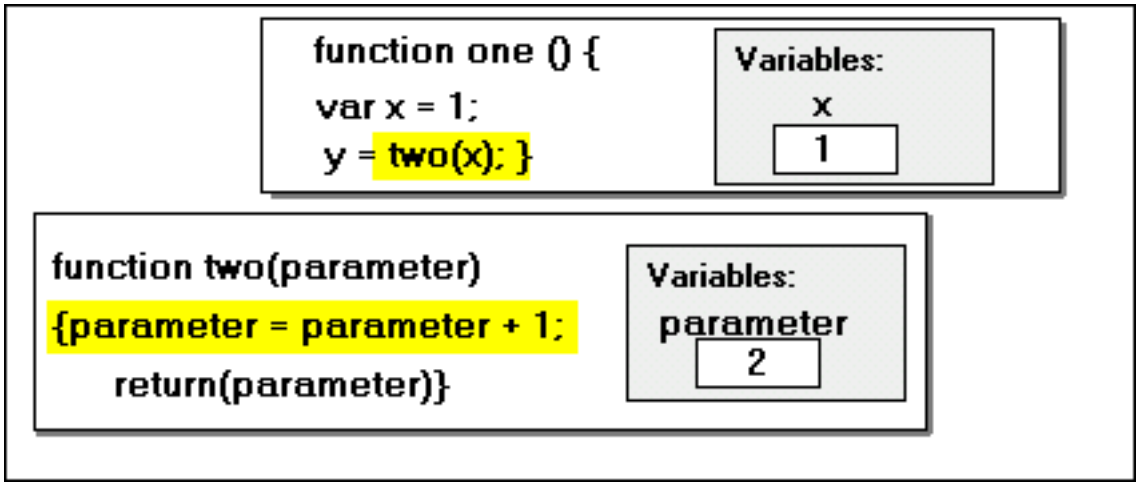
3.3 Data Exchange between Procedures

When a software system functionality is decomposed into a number of functional modules, data exchange/flow becomes a key issue. Imperative (Procedural) Programming Paradigm extends the concept of variables to be used as such data exchange mechanism.
Thus, each procedure may have a number of special variables called ***parameters***. The parameters are just named place-holders which will be replaced with particular values (or references to existing values) of arguments when the procedure is called.

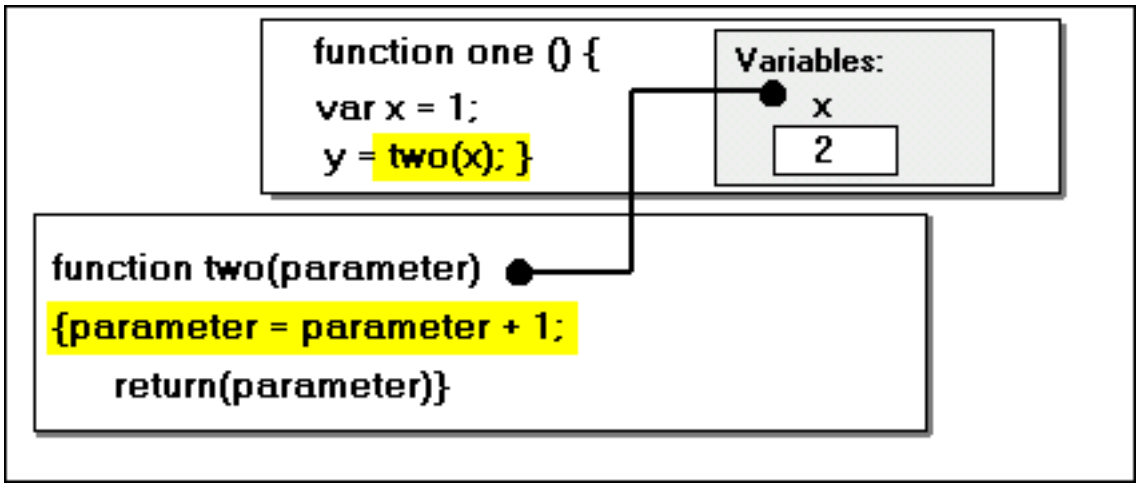


There might be two different techniques for such replacement which are known as: ***passing an argument value*** and ***passing an argument reference***.

In case of ***passing a value***, a current argument value is duplicated as a value for new parameter variable dynamically created for the procedure. In this case, variables used as arguments for calling sub-routines cannot be modified by imperative operators inside of the sub-routines.

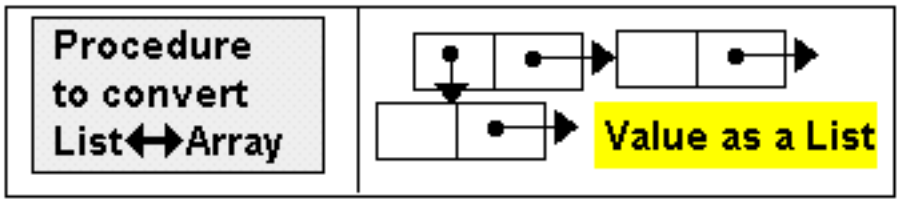


In case of ***passing a reference***, the sub-routine gets control (i.e. reference) to a current value of the argument variable. In this case, variables used as arguments for calling sub-routines can be modified by imperative operators inside of the sub-routines.



Thus, types of variables defined as parameters of a function should be equivalent to (or at least compatible with) types of variables (constants) used as arguments.

When strong static typing is enforced it can be difficult to write generic algorithms - functions that can act on a range of different types. ***Polymorphism*** allows "any" to be included in the type system.

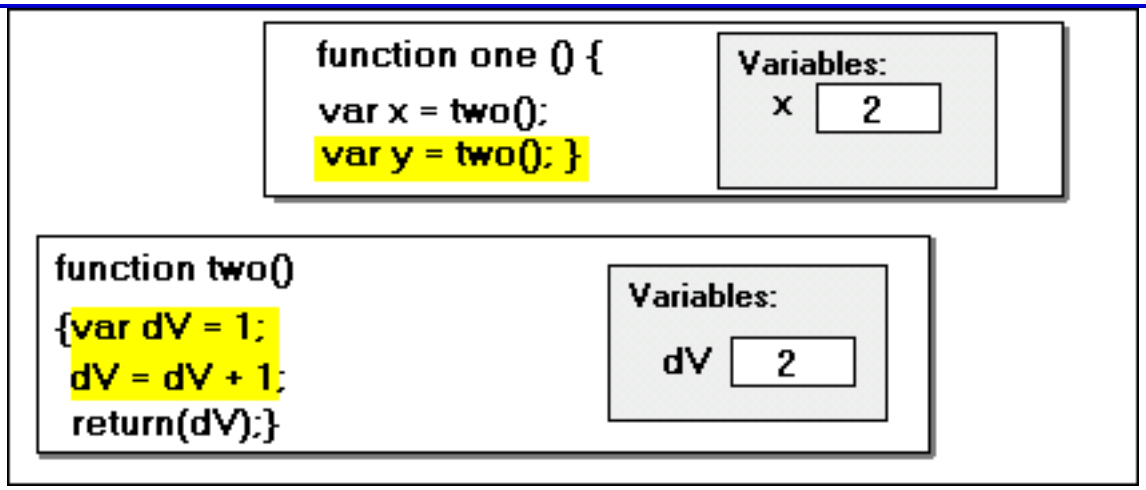



```
var A = new Array (1,2,3,4);
var y = SumList(A);
```

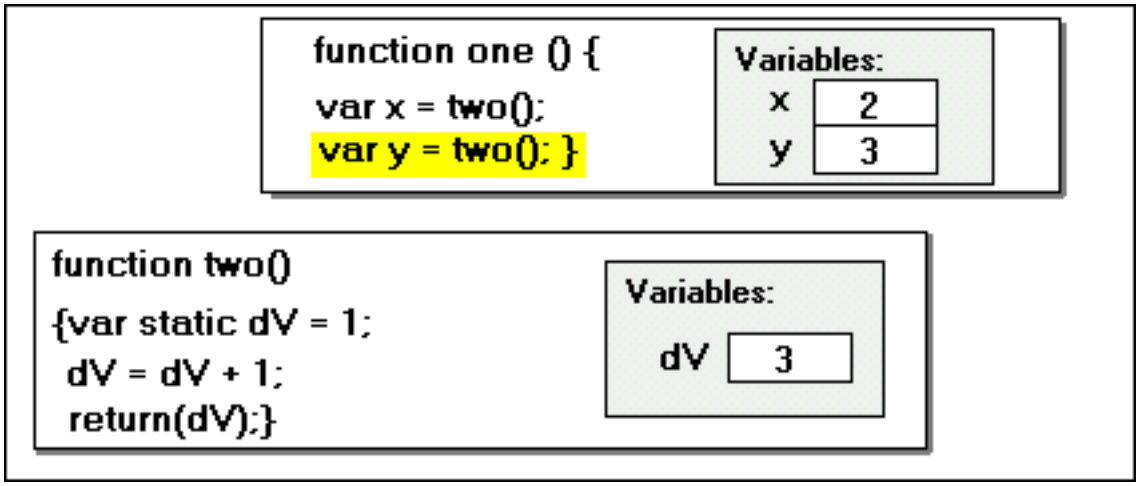
```
Lambda IX SumList
COND{
{list = nill, 0}
{true(), car(IX) + sumList(CDR(IX))}}
```

Pragmatically speaking, polymorphic languages allow to define new types as hidden functions which should be automatically applied to values of such "user-defined type" to convert it to values of a "standard" language type.

Normally, variables that are defined within a function, are created each time the function is used and destroyed again when the function ends. It is not possible to assign a value to the variable inside the function definition from outside.
Such variables are called ***dynamic local variables***.



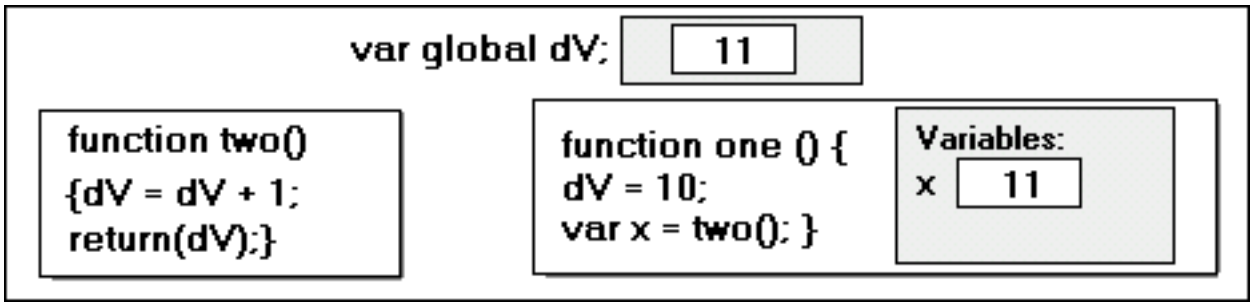
There may be also so-called ***static local variables***. Static local variables that are defined within a function, are created only once when the function is used for a first time. The value of such variable is not destroyed and can be reused when the function is called again.



Note that the function returns different values for one and the same set of arguments. Such functions are called ***reactive functions***.

Generally, testing and maintenance of projects having many reactive functions becomes a very difficult task. For practical reasons many software projects do use some static data. Note, it is still not possible to assign a value to the local static variable inside a function from outside.

There may be also so-called ***static global variables***. Static global variables that are defined within any function, are created only once when the whole software system is initiated. The value of such variable is never destroyed and can be reused by imperative operators inside any function.



Here, the function also demonstrates a "reactive" behavior. Maintaining and testing of projects heavily based on global variables becomes even more difficult than in case of local static variables. Nevertheless, for practical reasons many software development paradigms do use such global static variables.

4 Procedural Paradigm: Discussion

- **scalability/modifiability**

Plus: if a programmer comes up with a better way to implement a module then he/she simply replace the code within the function. Provided the interface remains the same - in other words the module name and the order and type of each parameter are unchanged - then no changes should be necessary in the rest of the application.
Minus: There is nothing to stop another programmer from meddling with the code within a module, perhaps to better adapt it to the needs of a particular application. There is also nothing to stop the code within the function making use of global variables, thus negating the benefits of a single interface providing a single point of entry.

- **integrability/reusability**
- **portability**
- **performance**
- **reliability**
- **ease of creation**

- **scalability/modifiability**
- **integrability/reusability**

Plus: anyone that needs a particular functionality can use an appropriate module, without having to code the algorithm from scratch.
One person can concentrate on writing a best possible module (function) for a particular task while others look after other areas.
Minus: modules have single interface providing a single point of entry, this makes an implementation of complex, reusable modules considerably difficult. For example, if we need to implement a common window interface (create a window, display a text, get modified text, etc.) we need a number of modules and common global variables. Usage global variables normally reduce integrability/reusability.

- **portability**
- **performance**
- **reliability**
- **ease of creation**

- **scalability/modifiability**
- **integrability/reusability**
- **portability**

Minus: since programmers essentially use variables, special data types,

input/output operations, percistent data (files) and user interface components specific for a particular platform, portability is not possible without considerable re-programming.

- **performance**
- **reliability**
- **ease of creation**

- **scalability/modifiability**
- **integrability/reusability**
- **portability**
- **performance**

Plus: since programmers normally use strong typing, special data types, and program in terms of operations performed by a computer, the paradigm allows to optimize a system performance on a very big extent.

- **reliability**
- **ease of creation**

- **scalability/modifiability**
- **integrability/reusability**
- **portability**
- **performance**
- **reliability**

Plus: due to strong typing and possible deterministic behaviour of all components (no static variables, no global variables), the paradigm allows to develop very reliable software systems.
Minus: the paradigm is not well suited for modifying an existing code (implementing a change request). Normally, any modification (especially introducing/modifying variables) requires a full testing phase to avoid bugs in other system components.

- **ease of creation**

- **scalability/modifiability**
- **integrability/reusability**
- **portability**
- **performance**
- **reliability**

- **ease of creation**

Minus: implementing a software system in terms of variables, imperative and control operators ia a very tedious job, and require a great deal of knowledge and experience.