Functional Decomposition
Top-Down Development

The **top-down** approach builds a system by **stepwise refinement**, starting with a definition of its abstract function. You start the process by expressing a topmost statement of this function, such as

“Translate a C prog to machine code”

and continue with a sequence of refinement steps. Each step must decrease the level of abstraction of the elements obtained; it decomposes every operation into a combination of one or more simpler operations.

“Read prog and produce seq of tokens”
“Parse tokens into abst syntax tree”
“Decorate tree with semantic info”
“Generate code from decorated tree”

The **top-down approach has a number of advantages:**

- logical, well-organized thought discipline
- can be taught effectively
- encourages orderly developmmt of systems
- helps the designer find a way through the apparent complexity that systems often present at the initial stages of their design
The **top-down** approach can indeed be useful for developing individual algorithms. But it also suffers from limitations that make it questionable as a tool for the design of entire systems:

- The very idea of characterizing a system by just one function is subject to doubt.

- By using as a basis for modular decomposition the properties that tend to change the most, the method fails to account for the evolutionary nature of software systems.

**Example: Payroll Program**

The system takes some inputs (record of hrs worked and employee info) and produces some outputs (paychecks . . .).

The top-down functional method is meant precisely for such well-defined problems, where the task is to perform a single function — the “top” of the system to be built.

Good systems have the detestable habit of giving their users plenty of ideas about all the other things they could do.

- Could it gather statistics on the side?

- We are going to start paying some employees monthly and others biweekly

- I need a summary every month for management, and one every quarter for the shareholders.

This phenomenon of having to add unanticipated functions to successful systems occurs in all application areas.
The new system is still, in many respects, “the same system” as the old one: still a payroll system; but the original “main function”, which may have seemed so important at first, often becomes just one of many functions; sometimes, it just vanishes, having outlived its usefulness.

If analysis and design have used a decomposition method based on the function, the system structure will follow from the designers’ original understanding of the system’s main function.

Each addition of a new function, however incremental it seems to the customer, risks invalidating the entire structure.

**It is crucial to find, as a criterion for decomposition, properties less volatile than the system’s main function.**

The **types of objects** manipulated by the system are more promising candidates.

Whatever happens to the payroll processing system, it likely will still manipulate objects representing employees, salary scales, company regulations, hours worked, pay checks.

Whatever happens to a compiler or other language processing tool, it likely will still manipulate source texts, token sequences, parse trees, abstract syntax trees, target code.

The argument only holds if we take a high-level enough view of objects. So the question of finding a suitably abstract description of objects is crucial.

To get a working implementation, you will of course, sooner or later, have to find out what it does.

**Better later than sooner**, says object-oriented wisdom. In this approach, the choice of main function is one of the very last steps to be taken in the process of system construction.
The developers will stay away, as long as possible, from the need to describe and implement the topmost function of the system.

Instead, they will analyze the types of objects of the system.

System design will progress through the successive improvements of their understanding of these object classes.

It is a **bottom-up** process of building robust and extendible solutions to parts of the problem,

and combining them into more and more powerful assemblies — until the final assembly which yields a solution of the original problem but, everyone hopes, is not the *only* possible one:

the same components, assembled differently and probably combined with others, should be general enough to yield as a byproduct, if you have applied the method well and enjoyed your share of good luck, solutions to future problems as well.

**OBJECT MOTTO**

> Ask not first what the system does:  
> Ask what it does it to!
Another unpleasant consequence of the top-down approach:

**premature binding of temporal relations**

You must specify the order of executing operations before you have had a chance to understand properly what these operations will do.

Object-oriented development, for its part, stays away from premature ordering.

The designer studies the various operations applicable to a certain kind of data, and specifies the effect of each, but defers for as long as possible specifying the operations’ order of execution.

This may be called the **shopping list approach**: list needed operations — all the operations that you may need; ignore their ordering constraints until as late as possible in the software construction process. The result is much more extendible architectures.

Working top-down means that you develop software elements in response to particular subspecifications encountered in the tree-like development of a system.

This approach is good at ensuring that the design will meet the initial specification, but it does not promote **reusability**.

Modules are developed in response to specific subproblems, and tend to be no more general than implied by their immediate context.

One can in principle include the concern for extendibility and generality in a top-down design process, and encourage developers to write modules that transcend the immediate needs which led to their development. But nothing in the method encourages generalization, and in practice it tends to produce modules with narrow specifications.

The very notion of top-down design suggests the reverse of **reusability**.
Designing for reusability means building components that are as general as possible, then combining them into systems.

This is a **bottom-up** process, at the opposite of the top-down idea of starting with the definition of “the problem” and deriving a solution through successive refinements.

**Top-down design: an assessment**

This discussion shows the method to be poorly adapted to the development of significant systems.

It remains a useful paradigm for small programs and individual algorithms; it is certainly a helpful technique to *describe* well-understood algorithms, especially in programming courses. But it does not scale up to large practical software.

By developing a system top-down you trade short-term convenience for long-term inflexibility; you unduly privilege one function over the others; you may be led to devoting your attention to interface characteristics at the expense of more fundamental properties; you lose sight of the data aspect; and you risk sacrificing reusability.
The Object Model

The major elements of the object oriented model are: abstractions, encapsulation, modularity, and hierarchy.

Abstractions

Abstractions arise from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences.

A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary.

An abstraction focuses on the outside view of an object, and so serves to separate an object's behavior from its implementation.

A concept qualifies as an abstraction only if it can be described, understood, and analyzed independently of the mechanism that will eventually be used to realize it.

Deciding upon the right set of abstractions for a given domain is the central problem of object-oriented design

Example:

On a hydroponics farm, plants are grown in a nutrient solution, without sand, gravel, or other soils. Maintaining the proper greenhouse environment is a delicate job, and depends upon the kind of plant being grown and its ages.

One must control diverse factors such as temperature, humidity, light, pH, and nutrient concentrations.

On a large farm, it is not unusual to have an automated system that constantly monitors and adjusts these elements. The purpose of an automated gardener is to efficiently carry out, with minimal human intervention, growing plans for the healthy production of multiple crops.
Encapsulation

The abstraction of an object should precede the decisions about its implementation. Once an implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients.

No part of a complex system should depend on the internal details of any other part.

Whereas abstraction helps people to think about what they are doing, encapsulation allows program changes to be reliably made with limited effort.

Encapsulation is the process of binding all of the details of an object that do not contribute to its essential characteristics. In practice one hides the representation of an object, as well as the implementation of its methods.

Abstraction and encapsulation are complementary concepts:

- **Abstraction** focuses upon the outside view of an object.
- **Encapsulation** prevents clients from seeing its inside view, where the behavior of the abstraction is implemented

Intelligent encapsulation localizes design decisions that are likely to change.

The ability to change the representation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.
Hierarchy

Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstractions than we can comprehend at one time.

Encapsulation helps manage this complexity by hiding the inside view of our abstractions.

Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough.

A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design, we greatly simplify our understanding of the problem.

Hierarchy is a ranking or ordering of abstractions.

Classes and Objects

Objects

An object represents an individual item, unit, or entity with a well defined role (behavior) in the problem domain.

An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class; the terms instance and object are interchangeable.

The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.

Behavior is how an object acts and reacts, in terms of its state changes and message passing. The behavior of an object is completely defined by its actions.

Identity is that property of an object which distinguishes it from all other objects.
Meaning of State

Example: Vending Machine

- What happens if the user first makes the selection and then puts the money?
- What happens if it says “Correct Change Only” and the user puts extra money?

In these examples we see how the behavior of an object is influenced by its history. The reason is the existence of state within the object.

Properties:
- Amount of money entered but not yet applied to a selection
- Change available
- Coins accepted
- Serial number

Example: Airplane, Elevator

The fact that every object has state implies that every object takes up some amount of space (memory).

The existence of state within an object means that the order in which operations are invoked is important. Thus, each object is like a tiny, independent machine (fsm).
The Meaning of Behavior

We characterize the behavior of an object by considering the operations that it may perform upon other objects. We call the entire set of operations that a client may perform upon an object its **protocol**.

A protocol denotes the ways in which an object may act and react, and thus constitutes the entire static and dynamic outside view of the abstraction.

**Example:** Chemical Process consisting of two valves used to control the input of liquids into a tank, a heater to control the temperature of the mix in the tank, and a valve to drain the tank.

Class

A class is a set of objects that share a common structure and a common behavior. A single object is simply an instance of a class.

The **interface** of a class provides its outside view and therefore emphasizes the abstraction while hiding its structure and the secrets of its behavior. This interface primarily consists of the declarations of all the operations applicable to instances of this class.

The **implementation** of a class is its inside view, which encompasses the secrets of its behavior. It consists primarily of the implementation of all of the operations defined in the interface.

The Role of Classes and Objects in Design

During analysis and the early stages of design, the developer has two primary tasks:

- Identify the classes and objects that form the vocabulary of the problem domain (key abstractions)
- Invent the structures whereby sets of objects work together to provide the behaviors that satisfy the requirements of the problem (mechanisms)
During this phase of development, the focus of the developer must be upon the outside view of these key abstractions and mechanisms.

In later stages of design and then moving into implementation, the task of the developer changes: the focus is on the inside view of these key abstractions and mechanisms, involving their physical representation.

CLASSIFICATION

Classification is the means whereby we order knowledge.

In OOD, recognizing the sameness among things allows us to expose the commonality within key abstractions and mechanisms, and eventually leads us to simple designs.

Unfortunately, there is no golden path to classification. The identification of classes and objects is the hardest part of OOD.

Why is classification so hard???

- There is no “perfect classification”, although certainly some classifications are better than others. Any classification is relative to the perspective of the observer.

- Intelligent classification requires a tremendous amount of creative insight. Sometimes the answer is evident, sometimes it’s a matter of taste, and at other times the selection of suitable components is a crucial point in the analysis.
In our experience, we identify classes and objects first according to the properties relevant to our particular domain. Such abstractions are usually easy, because they are directly part of the vocabulary of our problem space.

Entities that have a given **property** or collection of properties in common form a category (classical categorization).

The particular properties that should be considered in a given situation are highly domain-specific.

**Ex:** Color of a car

If this approach fails to yield a satisfactory class structure, then we next consider clustering objects according to some **conceptual** descriptions (conceptual clustering).

**Ex:** Love songs

If either of these two fails to capture our understanding of the problem domain, then we consider classification by association, through which clusters of objects are defined according to how closely each resembles some **prototypical** object (prototype theory). There are some abstractions that have neither clearly bounded properties nor concepts.

**Ex:** Games