Till now we have seen the data-abstraction barriers as powerful tools for controlling complexity.

By isolating the underlying representations of data entities, we can divide the task of designing a large program into smaller tasks that can be performed separately.

Till now we have seen the data-abstraction barriers as powerful tools for controlling complexity.

By isolating the underlying representations of data entities, we can divide the task of designing a large program into smaller tasks that can be performed separately.

But this kind of data abstraction is not yet powerful enough, because it may not always make sense to speak of "the underlying representation" for a data entity.

There might be more than one useful representation for a data entity, and we might like to design systems that can deal with multiple representations.

Certain choices while collecting data might be subjective and can have multiple correct choices.

- Complex numbers may be represented in two almost equivalent ways: in rectangular form (real and imaginary parts) and in polar form (magnitude and angle).

- A mobile phone company might use Aadhar number as the primary key to identify an individual, whereas the Traffic department might use the License number.

  Police department might need access to data from both these departments and their data bases might have different unique identifiers to refer to the same individual.
  When police department searches for an individual, the action it is performing is same (i.e fetch individual's info), but the data is stored in two different ways across the two platforms.

In practice, large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems *additively*, i.e. , without having to redesign or reimplement these modules.

In practice, large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems *additively*, i.e. , without having to redesign or reimplement these modules.

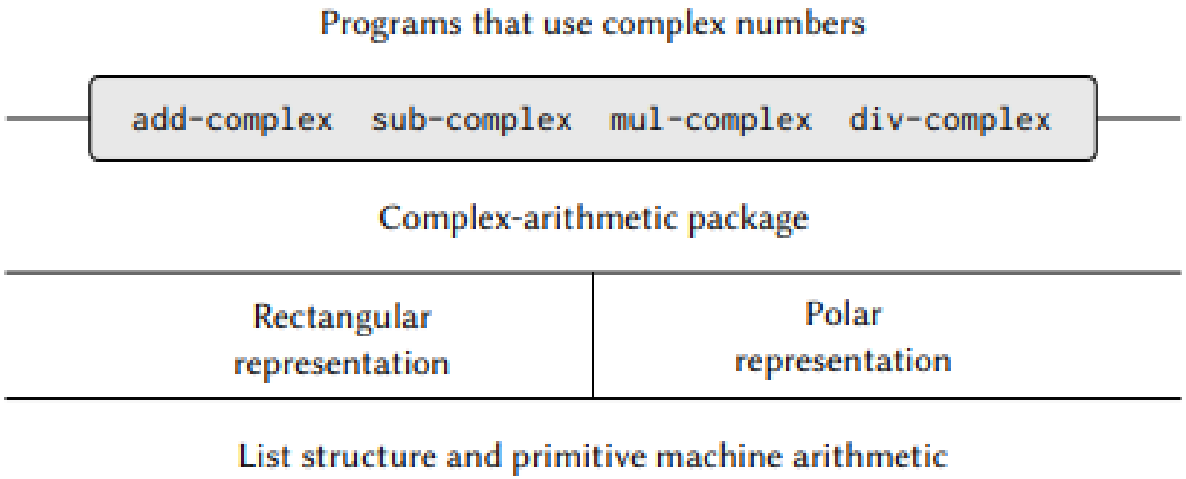
To mitigate this situation, one might construct *generic procedures*. These procedures can operate on data that may be represented in more than one way.

In practice, large programs are often created by combining pre-existing modules that were designed in isolation, we need conventions that permit programmers to incorporate modules into larger systems *additively*, i.e. , without having to redesign or reimplement these modules.

To mitigate this situation, one might construct *generic procedures*. These procedures can operate on data that may be represented in more than one way.

Few main techniques for building generic procedures are

- Type-tags
- Data-directed programming.



One might need a "vertical" barrier in order to separately design and install alternative representations.

There are two widely used representations for complex numbers as ordered pairs: rectangular form (real part and imaginary part) and polar form (magnitude and angle).

$$\begin{aligned}\text{Real-part}(z_1 + z_2) &= \text{Real-part}(z_1) + \text{Real-part}(z_2),\\ \text{Imaginary-part}(z_1 + z_2) &= \text{Imaginary-part}(z_1) + \text{Imaginary-part}(z_2).\end{aligned}$$

The rectangular form serves better if one performs more additions that multiplications, whereas polar form serves better when one performs more multiplications.

$$\begin{aligned}\text{Magnitude}(z_1 \cdot z_2) &= \text{Magnitude}(z_1) \cdot \text{Magnitude}(z_2),\\ \text{Angle}(z_1 \cdot z_2) &= \text{Angle}(z_1) + \text{Angle}(z_2).\end{aligned}$$

From the viewpoint of someone writing a program that uses complex numbers, the principle of data abstraction suggests that all the operations for manipulating complex numbers should be available regardless of which representation is used by the computer.

If Pablo and Roberto use two different representations for complex numbers they would provide the following set of constructors and selectors.

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z))
           (square (imag-part z)))))
(define (angle z)
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
```

Rectangular

```
(define (real-part z) (* (magnitude z) (cos (angle z))))
(define (imag-part z) (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (make-from-mag-ang r a) (cons r a))
```

Polar

The discipline of data abstraction ensures that the same implementation of add-complex, sub-complex, mul-complex, and div-complex will work with either of the representations.

Although Pablo and Roberto's representations are both equally good, there is a catch. **Once you choose a representation you would have to commit to that representation**.

Data abstraction can be viewed as an application of the "principle of least commitment".

Although Pablo and Roberto's representations are both equally good, there is a catch. **Once you choose a representation you would have to commit to that representation**.

Data abstraction can be viewed as an application of the "principle of least commitment".

Otherwise, if both representations are included in a single system, we will need some way to distinguish data in polar/rectangular form.

Issue:- If we want to find the magnitude of the complex number (3, 4), the answer could be 5 or 3 depending on whether the complex number is in rectangular or polar form.

Although Pablo and Roberto's representations are both equally good, there is a catch. **Once you choose a representation you would have to commit to that representation**.

Data abstraction can be viewed as an application of the "principle of least commitment".

Otherwise, if both representations are included in a single system, we will need some way to distinguish data in polar/rectangular form.

Issue:- If we want to find the magnitude of the complex number (3, 4), the answer could be 5 or 3 depending on whether the complex number is in rectangular or polar form.

Solution:- Include a *type tag*, the symbol rectangular/polar as part of each complex number.

In order to manipulate tagged data, we will assume that we have procedures

1. type-tag:- extract the tag from a data object
2. contents:- extract the actual contents (in this case polar or rectangular coordinates)
3. attach-tag:- takes a tag and contents and produces a tagged data object.

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))


(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum: TYPE-TAG" datum)))

(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum: CONTENTS" datum)))
```

With type tags, two different representations can coexist in the same system.
However, Pablo and Roberto must make sure that the names of their procedures do not conflict.

With type tags, two different representations can coexist in the same system.
However, Pablo and Roberto must make sure that the names of their procedures do not conflict.

One way to do this is for Pablo(Roberto) to append the suffix 'polar'('rectangular') to the name of each of his/her representation procedures.

So, in Pablo's (Roberto's) polar(rectangular) representation , the procedure real-part
would become real-part-polar (real-part-rectangular).

Additionally, the constructors also need to attach tags while creating a complex number. For example, in case of Roberto

```
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))
```

Each generic selector is implemented as a procedure that checks the tag of its argument and calls the appropriate procedure for handling data of that type.

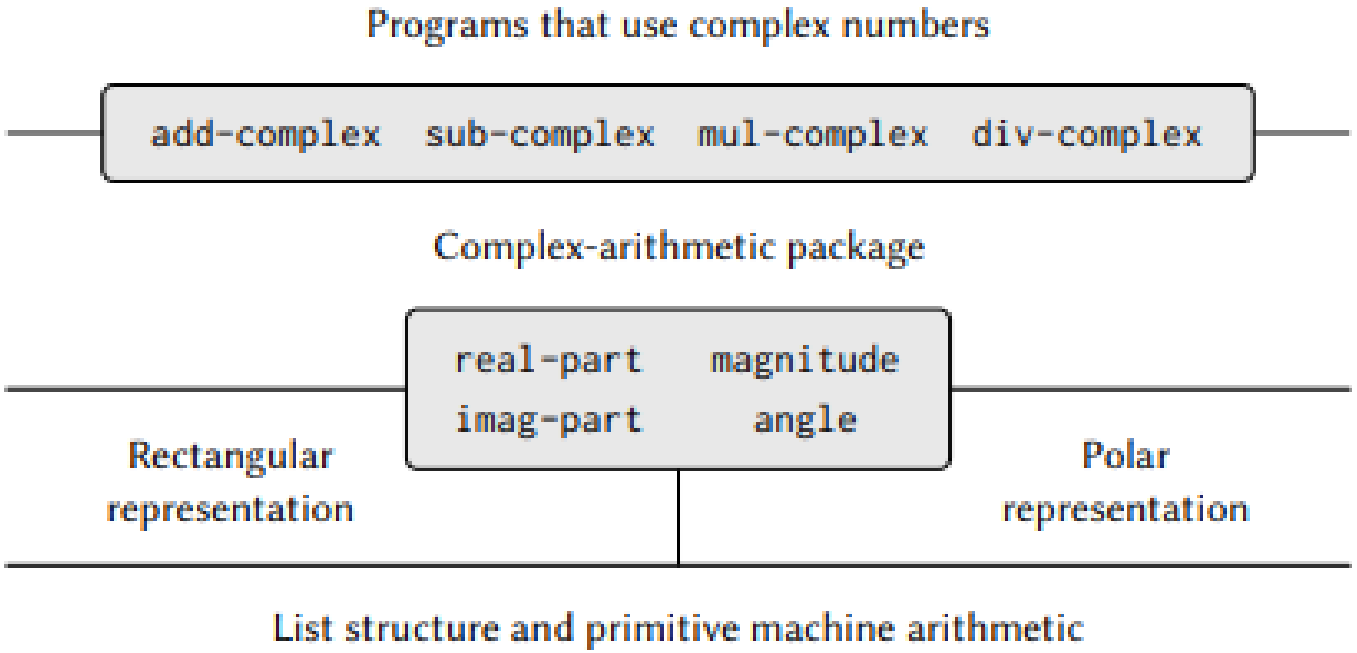
Each generic selector is implemented as a procedure that checks the tag of its argument and calls the appropriate procedure for handling data of that type.

For example, to obtain the real part of a complex number, real-part examines the tag to determine whether to use
Roberto's real-part-rectangular or
Pablo's real-part-polar.

In either case, we use contents to extract the bare, untagged datum and send this to the rectangular or polar procedure as required.

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type: REAL-PART" z))))
```

The complex-number arithmetic operations can use the previously defined procedures add-complex, sub-complex, mul-complex, and div-complex.



This is possible, because the selectors these procedures call are generic and so will work with either representation.

The previous solution for implementing the dispatch has two significant weaknesses.

1. Generic interface procedures (real-part, imag-part, magnitude, and angle) must know about all the different representations.

   If we wanted to add a new representation for complex numbers into our complex-number system, then
   - we would need to identify this new representation with a type,
   - add a clause to each of the generic interface procedures to check for the new type and apply the appropriate selector for that representation.

2. We must guarantee that no two procedures in the entire system have the same name.

   This is why Roberto and Pablo had to change the names of their original procedures.

The issue underlying both of these weaknesses is that the technique for implementing generic interfaces is not *additive*.

- People implementing the generic selector procedures must modify those procedures each time a new representation is installed.
- People interfacing the individual representations must modify their code to avoid name conflicts.

Observe that whenever we deal with a set of generic operations that are common to a set of different types we are, in effect, dealing with a two-dimensional table that contains the possible operations on one axis and the possible types on the other axis.

		Types	
		Polar	Rectangular
Operations	real-part	real-part-polar	real-part-rectangular
	imag-part	imag-part-polar	imag-part-rectangular
	magnitude	magnitude-polar	magnitude-rectangular
	angle	angle-polar	angle-rectangular

Data-directed programming is the technique of designing programs to work with such a table directly.

Previously, we implemented the mechanism that interfaces the complex-arithmetic code with the two representation packages as a set of procedures that each perform an explicit dispatch on type.

Data-directed programming is the technique of designing programs to work with such a table directly.

Previously, we implemented the mechanism that interfaces the complex-arithmetic code with the two representation packages as a set of procedures that each perform an explicit dispatch on type.

New solution:-
Implement the interface as a single procedure that looks up the combination of the operation name and argument type in the table to find the correct procedure to apply, and then applies it to the contents of the argument.

Adding a new representation package to the system

- would not require any changes to the existing procedures.
- would require adding new entries to the table.

To implement this plan, assume that we have two procedures, put and get, for manipulating the operation-and-type table:

- (put ⟨op⟩ ⟨type⟩ ⟨item⟩)
  inserts the ⟨item⟩ in the table, indexed by the ⟨op⟩ and the ⟨type⟩.
- (get ⟨op⟩ ⟨type⟩)
  looks up the ⟨op⟩, ⟨type⟩ entry in the table and returns the item found there. If no item is found, get returns false.

Roberto, who developed the rectangular representation, implements his code just as he did originally.

He defines a collection of procedures, or a *package*, and interfaces these to the rest of the system by adding entries to the table that tell the system how to operate on rectangular numbers.

- The internal procedures are the same procedures that Roberto wrote in isolation.
- Since these procedure definitions are internal to the installation procedure, Roberto needn't worry about name conflicts with other procedures outside the rectangular package.
- To interface these to the rest of the system, Roberto installs his real-part procedure under the operation name *real-part* and the type (rectangular), and similarly for the other selectors and constructors.

The complex-arithmetic selectors access the table by means of a general "operation" procedure called apply-generic, which applies a generic operation to some arguments.
apply-generic looks in the table under the name of the operation and the types of the arguments and applies the resulting procedure if one is present:

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error "No method for these types: APPLY-GENERIC"
                 (list op type-tags))))))
```

Using apply-generic, we can define our generic selectors as follows:

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

These selectors do not change at all if a new representation is added to the system.

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))
(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

We can also extract from the table the constructors to be used by the programs external to the packages in making complex numbers from real and imaginary parts and from magnitudes and angles.

An alternative implementation strategy is to decompose the table into columns and, instead of using "intelligent operations" that dispatch on data types, to work with "intelligent data objects" that dispatch on operation names.

We can do this by arranging things so that a data object, such as a rectangular number, is represented as a procedure that takes as input the required operation name and performs the operation indicated

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude) (sqrt (+ (square x) (square y))))
          ((eq? op 'angle) (atan y x))
          (else (error "Unknown op: MAKE-FROM-REAL-IMAG" op))))
  dispatch)
```

The corresponding apply-generic procedure, which applies a generic operation to an argument, now simply feeds the operation's name to the data object and lets the object do the work.

```
(define (apply-generic op arg) (arg op))
```

Note that the value returned by make-from-real-imag is a procedure—the internal dispatch procedure.

This style of programming is called message passing.