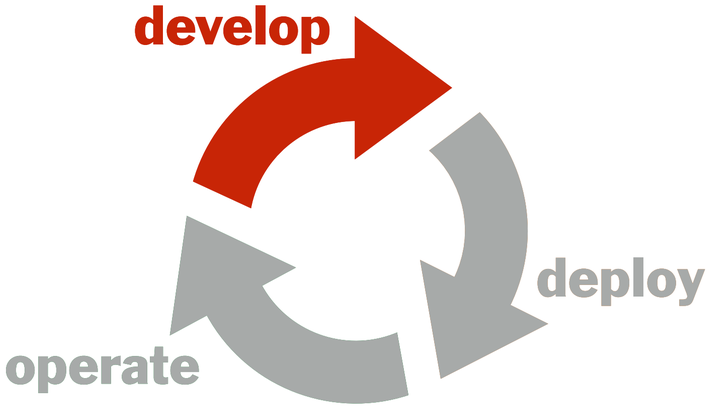
programming paradigms

learning objectives



- learn about the concept of programming paradigm
- learn how programming paradigms differ
- learn about object-oriented programming
- learn about functional programming

what's a paradigm?

a cognitive framework containing the basic assumptions, ways of thinking, and methodology that are commonly accepted by members of a discipline

a model of the world or in a more restrictive way of a part of the world, in our case of the world of programming

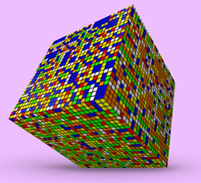
key programming paradigms

imperative

procedural programming 

object-oriented programming

declarative

functional programming 

logic programming

key programming paradigms

imperative

procedural programming

program = data structures + algorithms

typical languages: algol, pascal

declarative

logic programming

program = rules + inference engine

typical languages: planner, prolog

logic programming
in prolog

facts

```
cat(tom) :- true.
```

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)       :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
```

queries

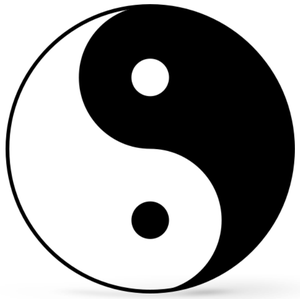
```
?- cat(tom).
Yes
```

```
?- cat(X).
X = tom
```

```
?- sibling(sally, erica).
Yes
```

```
?- father_child(tom, X).
X = sally
X = erica
```

try *swish.swi-prolog.org*



procedural programming in pascal

data structure

```
type
    String25 = String[25];
    Book = record
        title, author, isbn : String25;
        price : Real;
    end;
```

algorithm

```
program Printing;
var myBook : Book;

procedure Print(theBook : Book);
begin
  Writeln('Here are the book details:');
  Writeln('Title:  ', theBook.title);
  Writeln('Author: ', theBook.author);
  Writeln('ISBN:   ', theBook.isbn);
  Writeln('Price:  ', theBook.price);
end;

begin { main program }
  myBook.Title  := 'La Modification';
  myBook.Author := 'Michel Butor';
  myBook.ISBN   := '978-27-07303-12-7';
  myBook.Price  := 15.9;
  Print(myBook)
end.
```

procedural programming

procedural programming was a response to the growing complexity of algorithms data structures

with the number of lines of code growing exponentially in software, there was a need for modularity, encapsulation and code reuse

Letters to the editor: go to statement considered harmful

Full Text: PDF

Author: Edsger W. Dijkstra Technological Univ., Eindhoven, The Netherlands

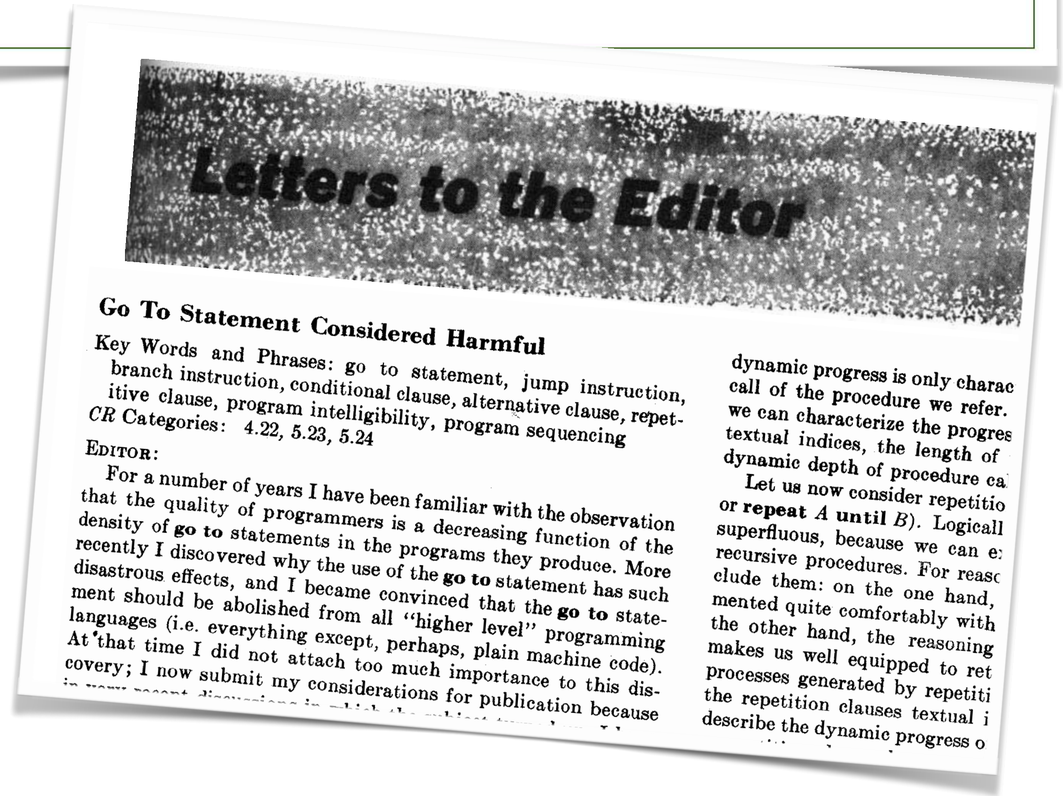
1968 Article

Published in:

· Magazine
Communications of the ACM CACM Homepage archive
Volume 11 Issue 3, March 1968
Pages 147-148
ACM New York, NY, USA
table of contents doi>10.1145/362929.362947

Bibliometrics

· Citation Count: 334
· Downloads (cumulative): 10,795
· Downloads (12 Months): 502
· Downloads (6 Weeks): 46



Letters to the Editor

Go To Statement Considered Harmful

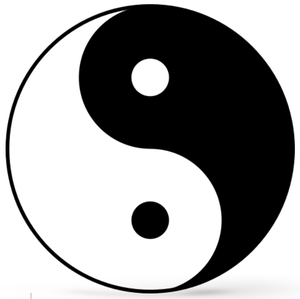
Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

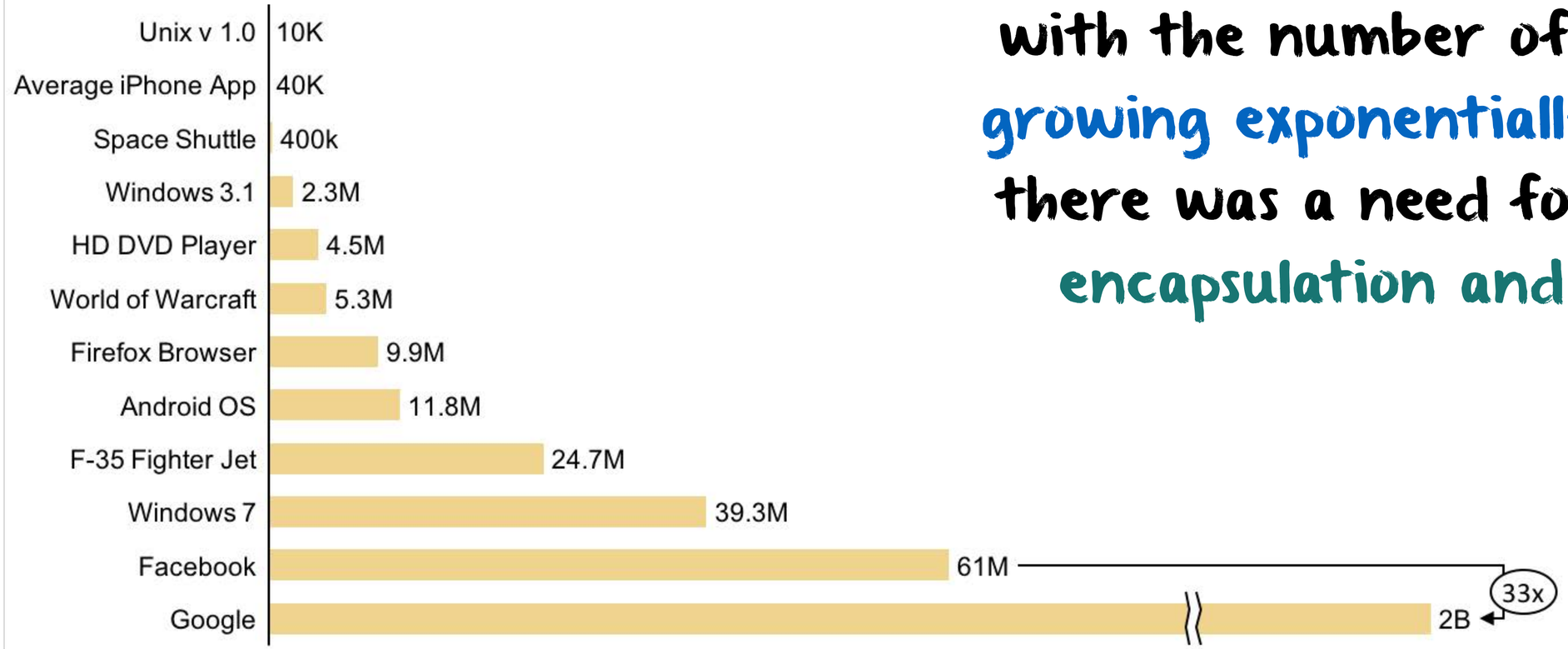
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because

dynamic progress is only charac
call of the procedure we refer.
we can characterize the progres
textual indices, the length of
dynamic depth of procedure ca

Let us now consider repetitio
or **repeat** *A* **until** *B*). Logicall
superfluous, because we can e:
recursive procedures. For reasc
clude them: on the one hand,
mented quite comfortably with
the other hand, the reasoning
makes us well equipped to ret
processes generated by repetiti
the repetition clauses textual i
describe the dynamic progress o

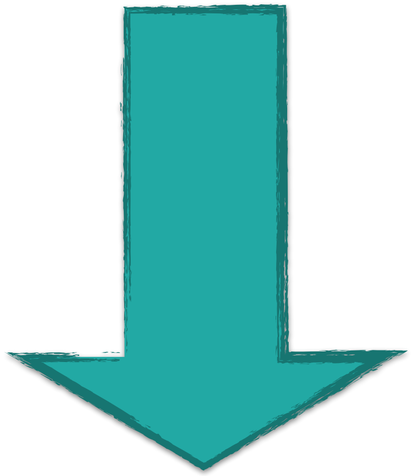


procedural programming



source: *thenextweb.com*

with the number of **lines of code** **growing exponentially** in software, there was a need for **modularity, encapsulation and code reuse**



object-oriented programming

object-oriented programming

objects and classes

an object represents particular "things" from the real world, or from some problem domain (e.g., "my blue rocking chair")

a class represents all objects of a given kind, e.g., "chairs"

object-oriented programming

objects and classes

many instances (objects) can be created from a single class

> the class can be seen as a kind of object factory (or a mold)

object-oriented programming

fields

the source code of classes defines the attributes (fields) and methods all objects of the class have

class Chair

```
String color;
String model;
boolean isBroken;
int age;
```

```
rotate (int angle)
```

instance myFirstChair	
color	"brown"
model	"wood"
isBroken	false
age	50

instance mySecondChair	
color	"green"
model	"shell"
isBroken	false
age	5

methods

methods may have parameters to pass additional information needed to execute it

```
chair.rotate(45)
```

object-oriented programming

specification vs implementation

what it does

how it does it

object-oriented programming

specification viewpoint

the viewpoint of someone simply wanting to use objects (not design them)

no need to know how objects are built to use them, only what can be done with them

encapsulation principle: allows us to hide (encapsulate) the complexity of objects

a class specifies the set of common behaviors offered by objects (instances) of that class

object-oriented programming
implementation viewpoint

the implementation viewpoint is concerned with how an object actually fulfills its specification (its contract)

the fields and methods define how the object will behave and are defined by its **class**

how it does it

object-oriented programming

specification vs implementation

```java
public class NumberDisplay {
    private int limit;
    private int value;

    public NumberDisplay(int limit, int value){
        this.limit = limit;
        this.value = value;
    }
    public NumberDisplay(int limit){
        this(limit, 0);
    }
    public int get() { return value; }
    public void set(int value) {
        this.value = value;
    }
    public void increment(){
        value = (value + 1) % limit;
    }
    public String toString(){
        if(value < 10) { return "0" + value; }
        else { return "" + value; }
    }
}
```

object-oriented programming
abstraction and modularization

abstraction is the ability to **ignore details** of parts to focus attention on a **higher level of a problem**

modularization consists in **dividing a complex object** into **elemental objects** that can be **developed independently**

the **encapsulation** offered by objects is the **cornerstone of modularization** because it hides implementation details

once **elemental objects** have been developed and tested, they **can be assembled** into a more complex object

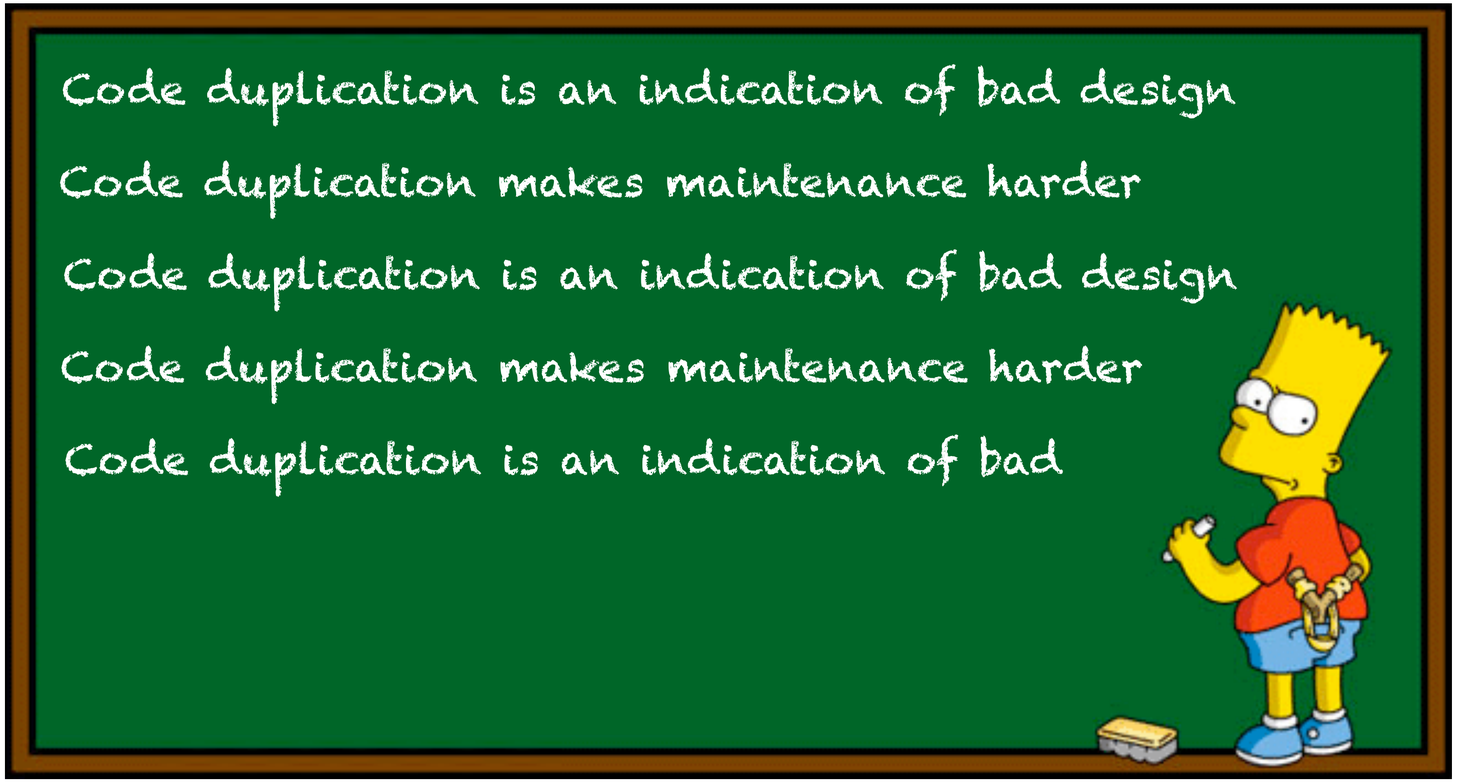
this is known as
code reuse

object-oriented programming

inheritance & polymorphism

object-oriented programming

inheritance & polymorphism



object-oriented programming

inheritance & polymorphism

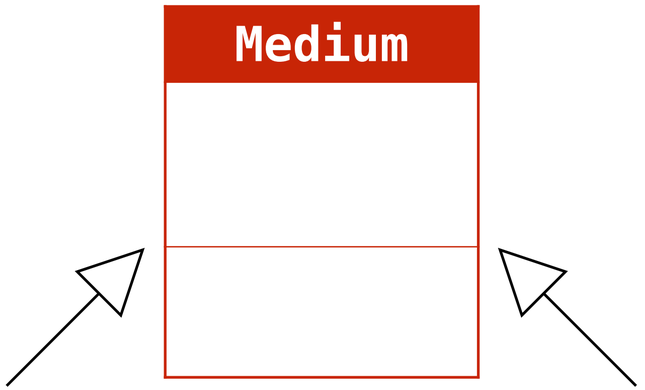
inheritance allows us to define one class, the subclass, as an extension of another, the superclass

a superclass defines common attributes

a subclass inherits all fields and methods from its superclass and defines specific attributes

object-oriented programming

inheritance & polymorphism



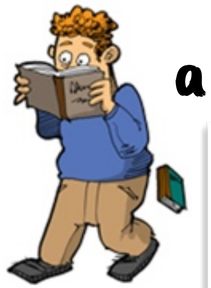
place common attributes in the superclass

object-oriented programming

inheritance & polymorphism

classes define types

superclasses define super types



a geek

a person with a devotion to something in a way that places him or her outside the mainstream*

*wikipedia

subclasses define subtypes

object-oriented programming

inheritance & polymorphism

substitution principle

objects of subtypes can be used where objects of supertypes are required

LINUX GEEK

CODE GEEK

KISS GEEK

object variables are polymorphic because they can hold objects of more than one type

object-oriented programming

inheritance & polymorphism

```
Book b = Book(title:"Hamlet", author:”Shakespeare”);
Vinyl v = Vinyl(title:"Abbey Road", artist:"Beatles", duration:47);
Medium m;
```

✘ `v = b`

✘ `b = v`

✔ `m = v`

✔ `m = b`

✘ `b = m`

`b = (Book)m;` ✔

the substitution principle only works...

this is type casting

object-oriented programming
code quality

coupling

coupling refers to links between separate units of a program

cohesion

cohesion refers to the number and diversity of tasks that a single unit is responsible for

a unit is either a class or a method

object-oriented programming
code quality

coupling

if two classes depend closely on many details of each other, we say they are tightly coupled

we aim for loose coupling

cohesion

if a unit corresponds to a single logical aspect, it has high cohesion

a class should represent one single, well defined entity

a method should be responsible for one and only one well defined task

we aim for high cohesion

object-oriented programming
code quality

loose coupling
makes it easy to

understand one class
without reading others

change a class without
affecting others

maintain the code
and make it **evolve**

high cohesion
makes it easy to

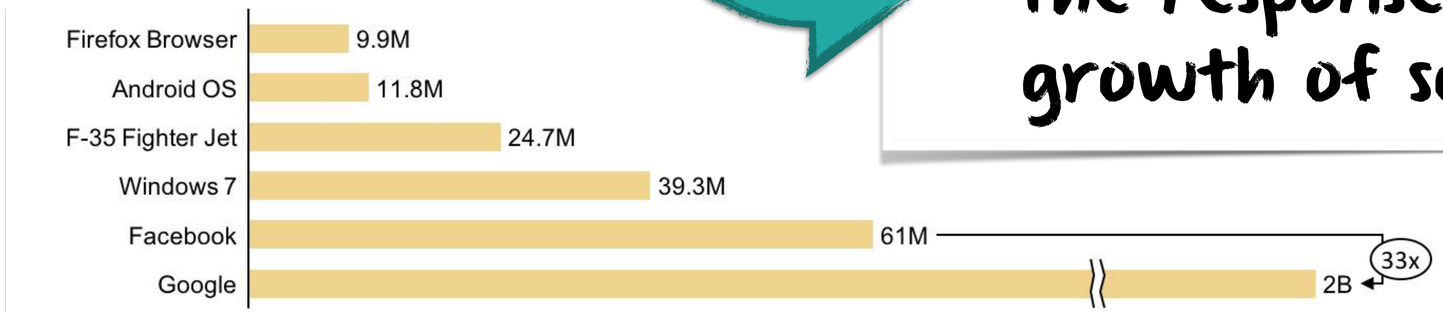
understand what a class or method
does and use descriptive names

avoid confusing the scope and
responsibility of a class or method

reuse the code of classes
or methods across projects

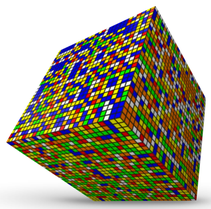
object-oriented programming

the response to the exponential growth of software code bases

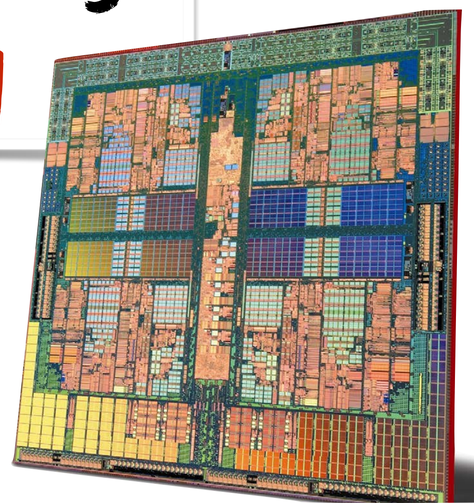


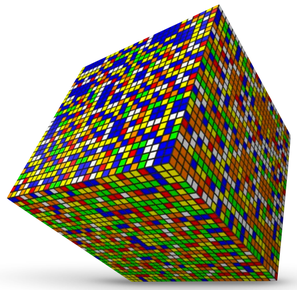
source: *thenextweb.com*

but with the massive increase in parallelism brought by hardware and operating systems, a new challenge emerged: how to safely manage concurrency



functional programming



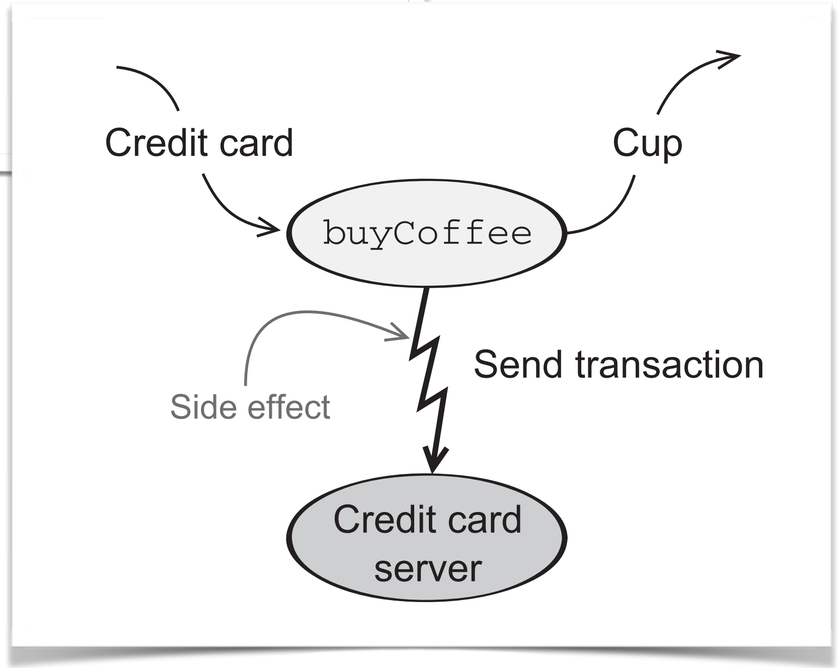


functional programming

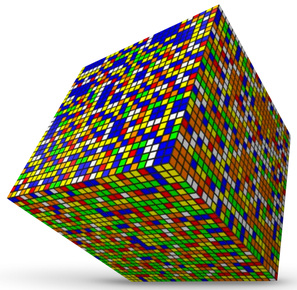
a function has a side effect
if it modifies something outside itself

charging a credit card modifies something outside `buyCoffee`, i.e., it requires contacting the credit card company

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    return cup
  }
}
```



this makes it difficult to test function `buyCoffee`



functional programming

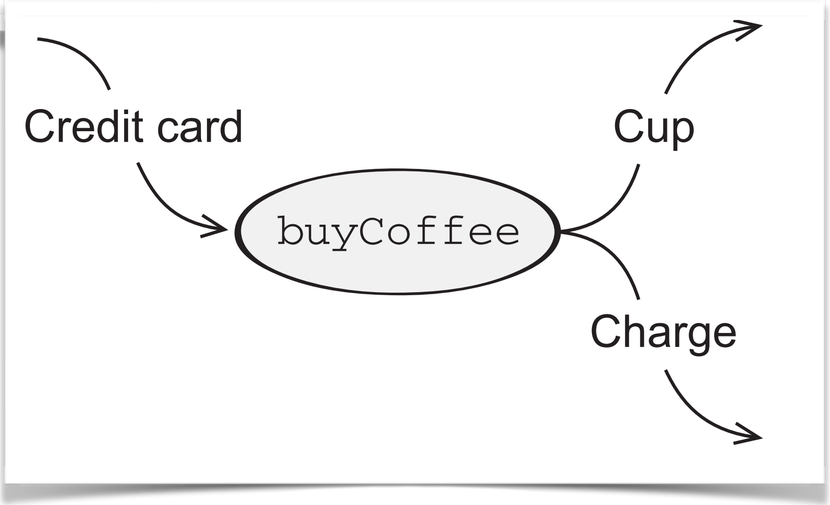
a pure function only return values and have no side effects

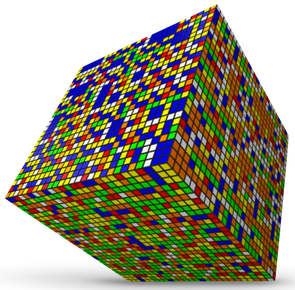
```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    return (cup,Charge(cc, cup.price))
  }
}

case class Charge(cc: CreditCard, amount: Double) {
  def combine(other: Charge): Charge =
    if (cc == other.cc)
      return Charge(cc, amount + other.amount)
    else throw new Exception("Charges on different credit cards cannot be combined")
}
```

now buyCoffee simply returns a tuple consisting of a Coffee and a Charge

class Charge reifies the action of charging the credit card and returns it as an object





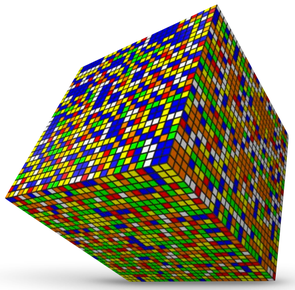
functional programming

referential transparency

an expression e is referentially transparent if, for all programs p, all occurrences of e in p can be substituted by the result of evaluating e without affecting the semantics of p

a function f is pure if the expression $f(x)$ is referentially transparent for all referentially transparent x

mathematical functions are by definition referentially transparent



functional programming

referential transparency

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = {
    val cup = new Coffee()
    cc.charge(cup.price)
    return cup
  }
}
```

```
...
val coffee = buyCoffee(myCreditCard)
...
```

⇎

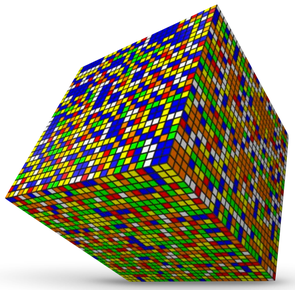
```
...
val coffee = new Coffee()
...
```

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    return (cup,Charge(cc, cup.price))
  }
}
```

```
...
val (coffee, charge) = buyCoffee(myCreditCard)
...
```

⇔

```
...
var cup = new Coffee()
val (coffee, charge) = (cup, Charge(myCreditCard, cup.price)
...
```



functional programming

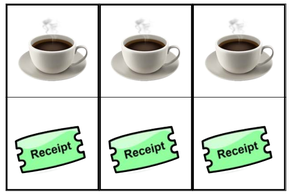
functional reuse and high-order functions

pure functions contribute to code reuse
because they can be easily composed

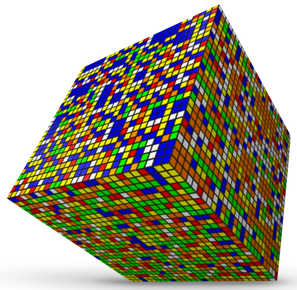
```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = { ... }

  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
    val (coffees, charges) = purchases.unzip
    return (coffees, charges.reduce((c1,c2) => c1.combine(c2)))
  }
}
```

assuming n = 3



what is this?



functional programming

functional reuse and high-order functions

a high-order function takes a function as parameter or returns function as its results

high-order functions are also called functionals or functors

```
class Cafe {
  def buyCoffee(cc: CreditCard): Coffee = { ... }

  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))
    val (coffees, charges) = purchases.unzip
    return (coffees, charges.reduce((c1,c2) => c1.combine(c2)))
  }
}
```

this is a parameter of type function
`(Charge,Charge) => Charge`

this concept comes from lambda calculus, a formal system in mathematical logic for expressing computation