## Goals of design

<table>
<thead>
<tr>
<th>Usable Software</th>
<th>Solves the problem, satisfies problem requirements, usable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Robust</td>
<td>Error free, resistant to incorrect input</td>
</tr>
<tr>
<td>Implementable</td>
<td>Minimize complexity</td>
</tr>
<tr>
<td>Understandable</td>
<td>Readable, not purposefully cryptic</td>
</tr>
<tr>
<td>Reusable</td>
<td></td>
</tr>
<tr>
<td>Generalizable</td>
<td>Solves related problems (parameters, generics)</td>
</tr>
<tr>
<td>Extendable</td>
<td>Can be modified to solve other problems</td>
</tr>
<tr>
<td>Others?</td>
<td></td>
</tr>
</tbody>
</table>
Abstraction, Inheritance, Polymorphism

Non OOP design's associated data and behavior from the perspective of the behaviors. Data was arguments of functions

```plaintext
function (data structure parameters)
sort (anArray)
```

OOP Abstraction groups related attributes (data) and/or methods (behaviors) into a syntactical concept (class, struct, property, generic, and to a lesser degree interface, delegate).

Inheritance allows ADTs to be factored with specific subclasses inheriting from more abstract superclasses -- type reuse and extension

Polymorphism allows specific behaviors (extended from more abstract Behaviors in specific subclasses) to be invoked in the more abstract (general / nameless) context.

Relaxes the coupling of the specification of the behavior from its usage. Minimizes errors, complexity, facilitates – reuse and generalization.
Encapsulation

Separate model implementation from programming interface (view)
  get / set accessor / mutator methods

C# Properties extends semantic constraints and hides the get / set
programming interface.

Usage is more transparent – "overloads" the assignment operator

Accessibility of ATD members: public, private, protected

Minimizes errors: fewer side effects and propagation of changes
"is - a"

abstract

"is - a"

specific

factor and "raise" common attributes and behaviors

= inherit use

Δ modify customize

+ add new behaviors

- hide (remove) behaviors
Composition (aggregation) represents the "has - a" relation.

If objectA needs to reference a behavior of objectB it does not have to inherit from objectA. it can have a reference attribute the accessess objectA "has - a" relations can be inherited (once established).

Object based languages (Modula, ADA) and non OOP languages (C, Pascal, Fortran) relied on composition for design.

Compare is-a and has-a … (usually more “has-a” than “is-a”)
Entertainment appliance - one box that has a monitor, TV, FM, AM receivers, VHS, DVD, and computer integrated subparts.) Separate components: monitor, TV, FM, AM receivers, VHS, DVD, and computer.
Interface

Inheritance is an "implementation inheritance".

The superclass's attributes and behaviors are written and usable in the subclass.

An abstract class can have attributes and can provide all, partial, or no implementation of behaviors.

Interfaces are a "definition inheritance" "mix ins"

Interfaces define in the "superclass" what has to be implemented in the subclass.

Implemented methods can be invoked in the interface's context.

Interfaces can not have attributes and provide no implementation of behaviors.

Interfaces are a "contract" between the "interface superclass" and the "implementor subclass" that the compiler can enforce.

Interfaces are a "must implement" subset of an "is-a" relationship.

An List<T> object is-an ennumerable object.
type membership

Class A

Class B

Class C

Class D

Interface A

Collections of

as Class A
aClass B
aClass C
as Interface A
aClass C
aClass D

object aClass C is-a Class A and is-a Interface A
## Containers / Collections

<table>
<thead>
<tr>
<th>Languages</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically typed (Smalltalk)</td>
<td>easy to define reusable classes</td>
<td>poor static error detection</td>
</tr>
<tr>
<td>Statically typed (C# Java)</td>
<td>good static error detection</td>
<td>typing complicates reusable abstractions</td>
</tr>
</tbody>
</table>

## Approaches

<table>
<thead>
<tr>
<th>Approaches</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>subclassing &amp; &amp; casting (Object ancestor)</td>
<td>works for most types in cast expression</td>
<td>specific type required in cast expression</td>
</tr>
<tr>
<td>subclassing &amp; &amp; method overriding (event interfaces)</td>
<td>no cast expression</td>
<td>methods must be known in advance</td>
</tr>
<tr>
<td>Templates</td>
<td></td>
<td>generics</td>
</tr>
</tbody>
</table>
element traversal

Subclass && cast
   Easy to add objects to collections ...
   Harder to work with, remove objects from collection ...

C# ArrayList and "as" operator
   foreach (AnObject tObj in anArrayList) {
      AnObject anObject = tObj as AnObject;
      if (anObject != null)
         // do something with anObject

Compare to Squeak / Smalltalk's visitor pattern with block
   aBag do: [ :i | 'do somthing with object i'].

C# generic

List<AnObject> list = new List<AnObject>();
   // add anObjects to list
list.Add(new AnObject(…));
...
   foreach (AnObject ao in list)
      Console.WriteLine(ao.ToString());
Correctness

Assertions (Eiffel, C#) enable testing of boolean conditions that should always be true at run (or debug) time.

```csharp
int IntegerDivide ( int dividend , int divisor ) {
    Debug.Assert ( divisor != 0 );
    return ( dividend / divisor );
}
```

Exception handling: try, catch, finally blocks

Use of "tested", "validated" class libraries

OOP design trade-off

Initially develop many small classes w/ "future usage" capabilities → assimilation / repackaging of classes → fewer classes in resulting design

Initially few large unfocused classes → factoring of classes → more "conceptually consistent" classes in resulting design
Patterns

repetitive pattern use reduce design complexity:
   Creation, Structural, Behavioral patterns (eg below)

Factories  construction of objects
Adapters   modify API of one class to the expected interface of a client -- "protocol stacks", translator
Chains     pass requests along a chain of receiving objects until the appropriate receiver responds. "ip to mac request"
Iterators  process collection of objects w/o knowing structure of collection
Mediator   promotes loose coupling by not having objects refer to each other directly (lookup table, behavior)
Observer   one to many dependency, one change is propagated to many
Visitor    operation to be applied to members of a collection
Design is a plan that is projected onto a product / application.

Implementation distorts the reflection of the initial (possibly ideal) design onto the actual application.

Software Engineering assumption:
↑ ideal of the design, ↓ implementation decisions, ↓ distortion
Can design be taught prior to, or separate from, implementation?

Study of technique / methods / media
(algorithms, data structures, programming languages) proceeds design.

Design patterns emerge from canonical works.

Do design tools shape designs?

Can design be practised? Does one become more efficient at design?

Is effort a constant?

\[
\text{program effort} = \text{design effort} + \text{implementation effort}
\]

Does an \( \uparrow \) in design effort cause a \( \downarrow \) in implementation effort?

Practise design
Architects design structures that can’t be built w/ current technology (eg. Paolo Soleri / Arcosonti)
Inheriting Software Metrics

Barnes, G.M. and Swim, B.R.,

Metrics enable software to be measured, software engineering assumptions to be experimentally tested.

Abstraction of Software Metrics, automation of metric computation ActQool Actor, "smalltalk", prototype extended class browser to provide metrics & visual representation. developed, gathered, reported correlations of OOP metrics
Qool Browser

/* Infix operation collection is eq
collections are
they cannot be eq
Def = {self, coll |
  if self ~ coll then
    if (class(self,
      cor ((idx :=
        then ^nil
        loop
          while (idx :=
            if not(self[
              then ^nil
            endif;
          )
        )
      )
    )
  )
*/

Design By Objectives:
- Quality variables [class]
- Quality methods [class]
- Functional Reqmts
- Attributes
- Availability
- Reliability
- Correctness
OOP Metrics

n = number of operators n1 + operands n2
N = frequency of operators N1 + operands N2

volume (V) = N (log₂ n)  .93
Length (N) = N1 + N2  .97  .99
V(G) ≈ number of predicates  .98  .98  .98

LOC  Volume  Length  V(G)  Self  Others  Methods

Self  .90  .95  .95  .92
Others  .82  .84  .83  .84  .72
Methods  .80  .67  .68  .66  .55  .46
Variables  .60  .62  .61  .71  .60  .59  .31

Self and Others are counts of messages sent to self or other methods
Methods and Variables are counts of local methods and variables in a class
LOC high correlation w/ all other metrics ∴ other metrics not needed