# OOP (C++): OO Modeling & Design

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iaşi, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2019/2020

# Plan

# A Kind of Definition

A model is an abstraction of the software system to be developed that makes the implementation (programming) of the system easier.

A model has three main characteristics (Herbert Stachowiak, 1973):

- *Mapping*: a model is always an image (mapping) of something.
- *Reduction*: a model does not capture all attributes of the original.
- *Pragmatism*: a model should be useful.

# Quality of a Model

It is determined by the following characteristics (Bran Selic, 2003):

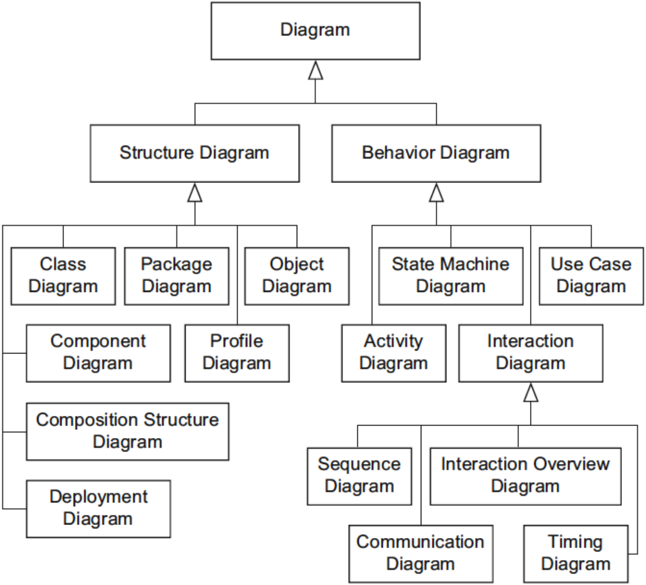
- Abstraction
- Understandability
- Accuracy
- Predictiveness
- Cost-effectiveness

# UML

- To write a model, you need a modeling language.
- UML (Unified Modeling Language) is a language and modeling technique suitable for object-oriented programming UML is used to view, specify, build, and document object-oriented systems
- In this course we will use UML elements to explain the concepts and laws of OOP.
- Free soft tools: BOUML, Argouml (open source), Visual Paradigm UML (Online Express Edition),. . .

# Plan

# UML Diagrams



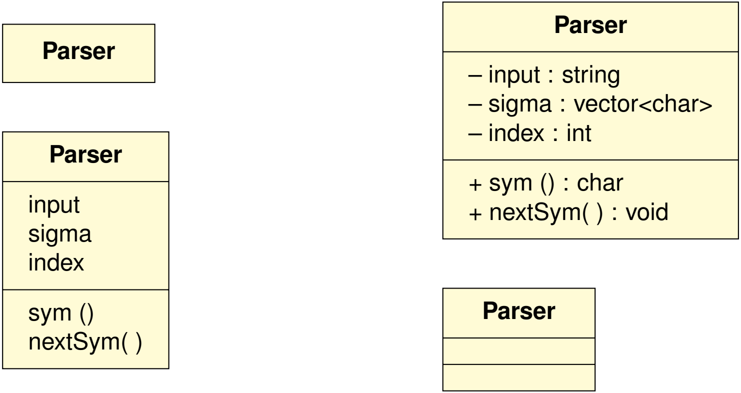
# Plan

# Specification of a Class in UML

```cpp
class Parser {
 private:
  string input;
  vector<char> sigma;
  int index;
 public:
  char sym();
  void nextSym();
  ...
};
```

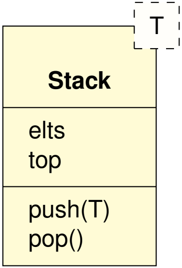
| **Parser** |
|---|
| – input : string<br>– sigma : vector<char><br>– index : int |
| + sym () : char<br>+ nextSym( ) : void |

# Notation


Parser
Parser
input
sigma
index
sym ()
nextSym( )
Parser
– input : string
– sigma : vector<char>
– index : int
+ sym () : char
+ nextSym( ) : void
Parser

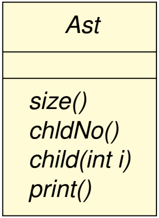
# Parametrized Class

```
template<class T>
class Stack {
 private:
  T* elts;
  int top;
 public:
  void push(T);
  void pop();
  ...
};
```



# Abstract Class

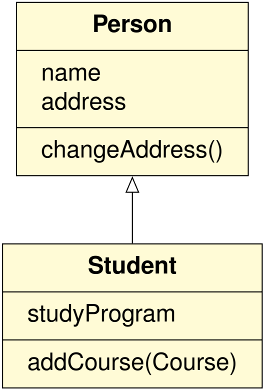
```
class Ast {
 public:
  virtual int size() = 0;
  virtual int chldNo() = 0;
  virtual Ast* child(int i) = 0;
  virtual void print() = 0;
};
```




Note the use of the italic font.

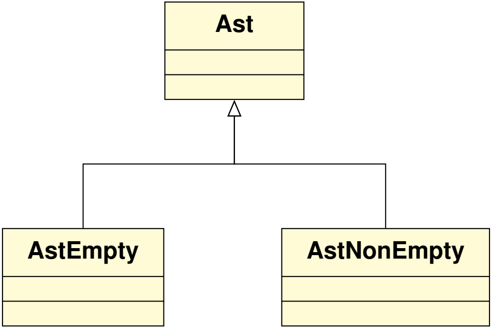
# Generalization: Inheritance

```
class Person {
  string name;
  string address;
 public:
  void changeAddress();
};
class Student : public Person {
  list<Course> studyProgram;
 public:
  void addCourse(Course);
};
```



# Generalization: Classification

```
class AstEmpty : public Ast {
  ...
};
class AstNonEmpty : public Ast {
  ...
};
```

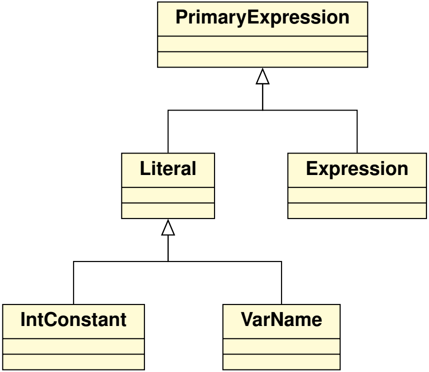


# Classification: A Simple Case Study

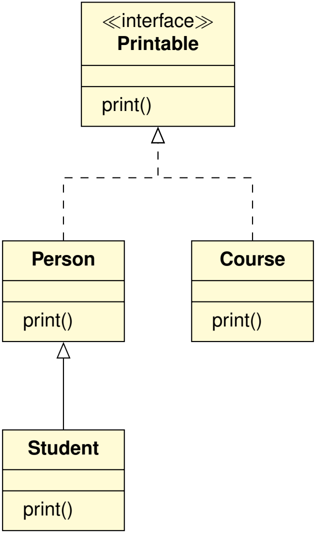
Frm C++ Grammar (simplified)

```
literal:
        int-constant
        var-name

primary-expression:
        literal
        ( expression )
```



# Interface



≪interface≫
Printable
print()
Person
print()
Course
print()
Student
print()

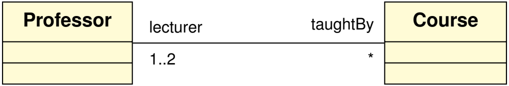
# Interfaces in C++

Are described using abstract classes:

```
class Printable {
 public: virtual void print() = 0;
};

class Person : public Printable {
 public: virtual void print();
}
...
void Person::print() {
  ...
}
```

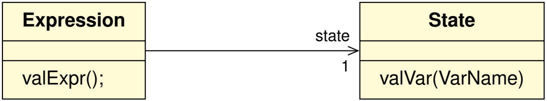
# Bidirectional Binary Associations




```
class Professor {
 private: list<Course*> taughtBy;
   ...
}
class Course {
  private: Profesor* lecturer[2];
   ..
}
```

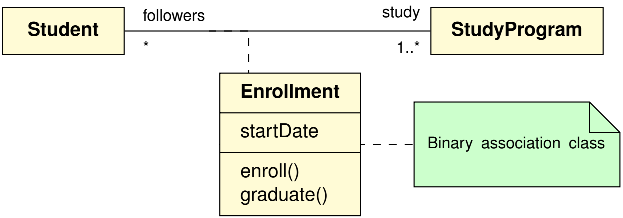
# Unidirectional Binary Associations

– an expression is evaluated in a state (the expression knows the state it is evaluated)
– the state does not know for which expression it is used for evaluation



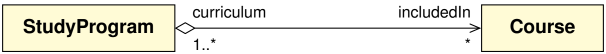

```
class Expression {
 private:  State* state;
 public: int valExpr(); // uses state->valVar()
   ...
}
class State {
  public : int valVar(VarName vn);
      // returns the value of the variable vn
   ...
}
```

# Association Class




```
class Enrollment {
  // class invariant: study != empty
 private:
  list<Student*>  followers;
  list<StudyProgram*> study;
  Date statDate;
public:
  void enroll(Student, StudyProgram);
  void graduate(Student, StudyProgram);
  ...
};
```

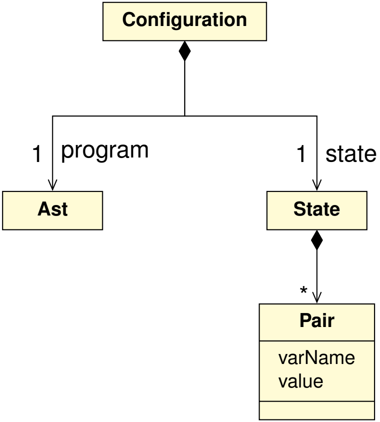
# Shared Agregation




```
class StudyProgram {
 private: vector<Course*> curricullum;
 ...
};\
class Course {
  // class invariant: includedIn != empty
 private: list<StudyProgram*> includedin;
 ...
};
```

# Strong Agregation (Composition)

configuration $=$ $\langle$ program, state $\rangle$
state $=$ var1 $\mapsto$ *val*1, var2 $\mapsto$ *val*2, $\ldots$




```
class Configuration {
 private:
  Ast program;
  State state;
  ...
};
class State {
 private:
  map<VarName, Value> _state;
 ...
};
```

# Plan

# Plan

# Definition

### Single-Responsibility Principle

A class should have one and only one reason to change, meaning that a class should have only one job.

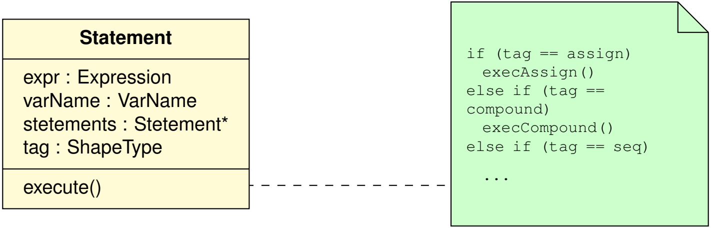
Robert C. Martin. Principles of Object Oriented Design.
https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view

# ~~SRP~~ 1/2

From C++ grammar (modified)

```
statement:
        var-name = expression ;  //asignement
        { statement }             // compound statetement
        (statement)*              //sequential composition
        if ( expression ) statement    // if
        if ( expression ) statement else statement  //if
        ...
```

# ~~SRP~~ 2/2




Too many responsibilities.

# SRP 1/2

From C++ grammar (simplified)

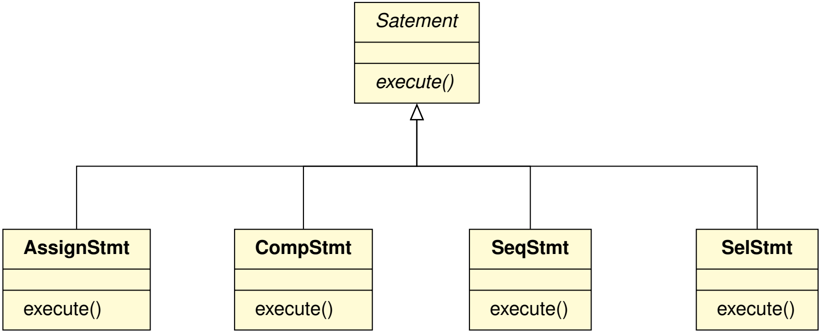
```
statement:
        assign-statement
        compound-statement
        statement-seq
        selection-statement

assign-statement:
        var-name = expression ;

compound-statement:
        { statement }
statement-seq:
        statement
        statement-seq statement

selection-statement:
        if ( condition ) statement
        if ( condition ) statement else statement
```

# SRP 2/2


Satement
execute()
AssignStmt
execute()
CompStmt
execute()
SeqStmt
execute()
SelStmt
execute()

# Plan

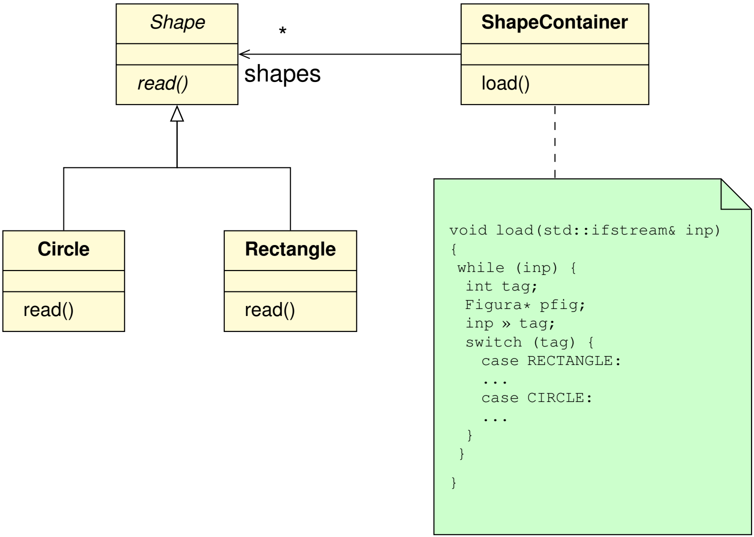
# Definition

## Open-Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Robert C. Martin. Principles of Object Oriented Design.
`https://drive.google.com/file/d/0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view`

# ~~OCP~~




If we add more shapes, we have to modify `ShapeContainer::load()`.

## OCP

The previous example can be fixed to respect OCP using Object Factory Pattern (next lecture).

# Plan

# Definition

## Liskov[1] Substitution Principle

Let $P(x)$ be a property provable about objects $x$ of type $T$. Then $P(y)$ should be true for objects $y$ of type $S$, where $S$ is a subtype of $T$.

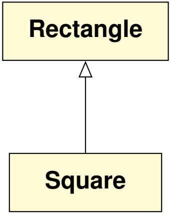
In other words, $S$ is a subtype of $T$ if anywhere you can use a $T$, you could also use an $S$ (a $T$ can be substituted by an $S$).

Barbara Liskov, Jeannette M. Wing: A Behavioral Notion of Subtyping. ACM Trans. Program. Lang. Syst. 16(6): 1811-1841 (1994)

[1] Turing Award, 2008, for contributions to programming languages development, especially OO languages.

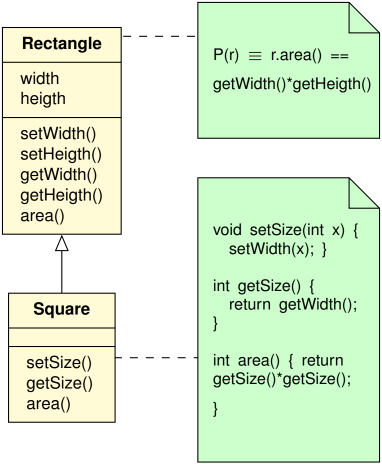
# A Counter-Example 1/4

A rushed modeling of the relation "Any square is a rectangle" (this is an instance of the "is a" relation):



# A Counter-Example 2/4

Let’s refine it:



# A Counter-Example 3/4

Consider the code:

```
Square s;
s.setWidth(5);        // inherited
s.setHeight(10);      // inherited
```

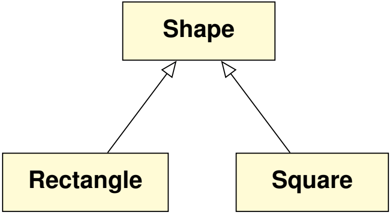
Assume that `s` plays the role of a `Rectangle`.
Does it satisfy the property P(`s`)? No.

**Question**

What is the right relationship between "subclass" and "subtype"?

# A Counter-Example 4/4

A correct hierarchy:



# Question

It seems that the subclass relation, given by

(child-class, partent-class)

is not the same with that of subtype.

So,
What is the right relationship between "subclass" and "subtype"?

# Types, Types, Types, . . . 1/3

From (online) manuals[2] (partial):
Objects, references, functions including function template specializations, and expressions have a property called type, which both restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits.

Type classification
The C++ type system consists of the following types:

- fundamental types
  - the type void
  - signed integer types int, long int, . . .
  - floating-point types float, double,. . .
  - . . .
- compound type
  - reference type
  - pointer type
  - array types
  - function types
  - class types
  - . . .

[2] https://en.cppreference.com/w/cpp/language/type

# Types, Types, Types, . . . 2/3

So, a class is a type?

No, not completely true.

A class includes two orthogonal things:

- static/behavioral type
- implementation

# Types, Types, Types, . . . 3/3

Static types (abstract view):

| Declaration of *X* | Type associated to *X* | Remark |
|---|---|---|
| `int X;` | `int` | |
| `struct X { int a; double b; }` | `int` × `double` | product type |
| `int X(double a);` | `double` → `int` | function type |
| `class X { int size; public: int getSize(); void setSize(int a); }` | `void` → `X`<br>`X` → `int`<br>`X` × `int` → `X` | constructor `X()`<br>`getSize`<br>`setSize` |

For classes, only the types of the public members (interface) are considered.

# Static Subtype 1/2

A type $T$ consists of:

- a set of possible values which a variable/expression can possess during program execution
- a set of operations/functions that can apply values

$S$ is a subtype of $T$, $S <: T$ if:

- $S$ values "are" $T$ values
- the operations/functions applied to $T$ can also be applied to $S$ preserving semantics

# Static Subtype 2/2

Examples: `int` <: `long int`

We do not have `int` <: `float` because _/_ on `int` has a different semantics from that of `float`.

For classes definition becomes more complex.

# On Contravariance/Covariance

Within a type system, the variance refers how the subtype relation is promoted to compound types.

- a rule is covariant if preserves the ordering of types $<:$
- a rule is contravariant if it reverses this ordering $<:$
- a rule is invariant if it is not neither covariant nor cotravariant

E.g., the rule for function types is contravariant on arguments and covariant on result:

$$\frac{A' <: A \quad B <: B'}{(A \rightarrow B) <: (A' \rightarrow B')}$$

C++ is invariant on parameters type and covariant on result type.

# Behavioral Type of a Class 1/3

Consider the following simple example:

```
class Square {
  int size;
 public:
  Square(int s = 0);
  int getSize();
  int area() throw(overflow_error);
  void halve();
}
```

# Behavioral Type of a Class 3/3

Includes also the behavioral interface of the objects and consists of:

- a description of the value space
  - ∗ class invariants
    Example: `getSize()` $\geq 0$
  - ∗ for each constructor $c$
    - – its signature
      Example: `Square` : `int` $\rightarrow$ `Square`
    - – precondition
      Example: `s` $\geq$ `0`
    - – postcondition
      Example: `getSize()` $\geq 0$
    - – the exceptions it signals

# Behavioral Type of a Class 2/3

- for each public method *m*:
  - ∗ its signature
    Example: $\texttt{area} : \texttt{Square} \to \texttt{int}$
    $\texttt{halve} : \texttt{Square} \to \texttt{Square}$
  - ∗ precondition
    Example: $\texttt{even}(\texttt{getSize}())$ (for $\texttt{halve}$)
  - ∗ postcondition
    Example: $\texttt{area}() == \texttt{0.25} * \texttt{old}(\texttt{area}())$ (for $\texttt{halve}$)
  - ∗ the exceptions it signals
- for each public attribute *a*:
  - ∗ its type

# A Possible Specification for the Behavioral Type

```
type Square {
    invariant getSize() >= 0
    Square(int s = 0)
      requires s >= 0
      ensures true
      noexcept(true);
    int getSize()
      requires true
      ensures true
      noexcept(true);
   int area()
      requires true
      ensures result == getSize()*getSize()
      throw(overflow_error)
   void halve();
      requires even(getSize())
      ensures area() == 0.25*old(area())
      noexcept(true);
}
```

# The Substitution Principle Explained using Behavioral Types 1/2

A child class (subclass) $B$ is a behavioral subtype of the parent class (superclass) $A$, $B <: A$, if:

- the invariants of the superclass are preserved by the subclass ($B$ values are $A$ values): the conjunction of the $B$ invariants implies conjunction of the $A$ invariants, i.e.,

$$\bigwedge_{\phi \in Inv(B)} \phi \implies \bigwedge_{\phi \in Inv(A)} \phi$$

- for each overridden method $m$
  - $B::m()$ and $A::m()$ have the same number of arguments
  - contravariance of arguments: if the $i$-th argument of $B::m()$ is $\beta_i$ and the $i$-th argument of $A::m()$ is $\alpha_i$, then $\alpha_i <: \beta_i$
  - covariance of result: if the result type of $A::M()$ is $\alpha$ and the result type of $B::m()$ is $\beta$, then $\alpha <: \beta$

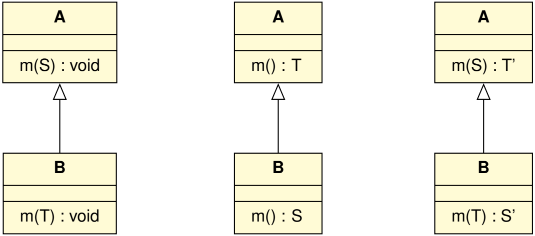
# The Substitution Principle Explained using Behavioral Types 2/2

- exception rule: the set of exceptions signaled by $B{::}m()$ is included in the set of exceptions signaled by $A{::}m()$
- semantics is preserved:
  - the precondition of $A{::}m()$ implies the precondition of $B{::}m()$
  - the postcondition of $B{::}m()$ implies the postcondition of $A{::}m()$

N.B. This is a simplified version of the original definition.
**Exercise**. Show that $T <: T$ ($<:$ is reflexive).

# Contravariance/Covariance Rule with Diagrams

$S <: T$
$S' <: T'$

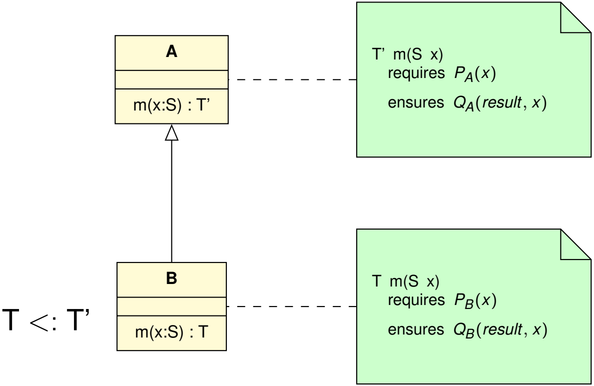



Contravariance of arguments

Covariance of result

Contravariance of arguments and Covariance of result

# Semantics Rule with Diagrams




$$P_B \implies P_A$$

$$Q_A \implies Q_B$$

# Contra-Example Revisited

The example "any square is a rectangle" violates several rules:

- the nethod `Square::setHeigth()` does not preserve the `Square` invariant $\texttt{getWidth}() = \texttt{getHeigth}()$
- the postcondition of `Square::area()` does not imply the postcondition of `Rectangle::area()`

# C++: (Contra-)Example with Contravariance of Arguments?

Not really:

```
class T { public: char t; T() { t = 'a'; } };
class S : public T { public: int s; S() { s = 1;} };

class A {
    public:
    virtual void m(S* s) { std::cout <<"\ns = " << s->s << "; ";}
};

class B : public A {
    public:
    void m(T* t) override { std:: cout << "\ns = 1; t = " << t->t << ";\n";}
};

int main(void) {
    S* s = new S;
    T* t = new T;
    A* pa = new A;      B* pb = new B;
    pa->m(s);            pb->m(s);
}
```

Output:

```
main.cpp:13:10: error: 'void B::m(T*)' marked 'override', but does not override
    void m(T* t) override { std:: cout << "\ns = 1; t = " << t->t << ";\n";}
         ^
```

# C++: (Contra-)Example with Covariance of Arguments?

```
class T { public: char t; T() { t = 'a'; } };
class S : public T { public: int s; S() { s = 1;} };

class A {
    public:
    virtual void m(T* t) { std::cout <<"\ns = " << t->t << "; ";}
};

class B : public A {
    public:
    void m(S* s) override {A::m(s); std::: cout << "t = " << s->s << ";\n";}
};

int main(void) {
    S* s = new S;
    T* t = new T;
    A* pa = new A;     B* pb = new B;
    pa->m(t);          pb->m(t);
}
```

Output:

```
main.cpp:13:8: error: 'void B::m(S*)' marked 'override', but does not override
   void m(S* s) override {A::m(s); std::: cout << "t = " << s->s << ";\n";}
        ^
```

Remember that C++ is invariant on arguments.

# Plan

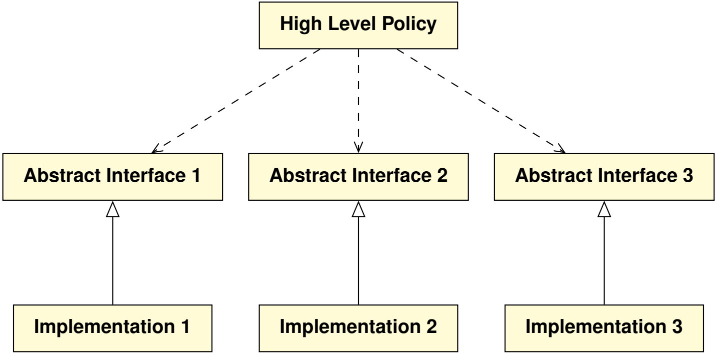
# Definition

- High-level modules should not depend on low-level modules. Both should depend on abstractions.

- Abstractions should not depend on details. Details should depend on abstractions.

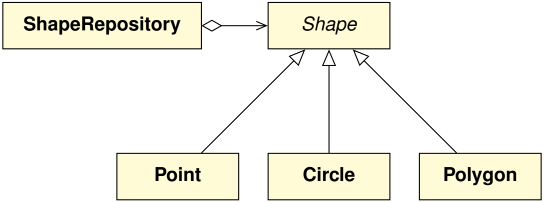
Robert C. Martin. Principles of Object Oriented Design.
https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf

# Abstract View

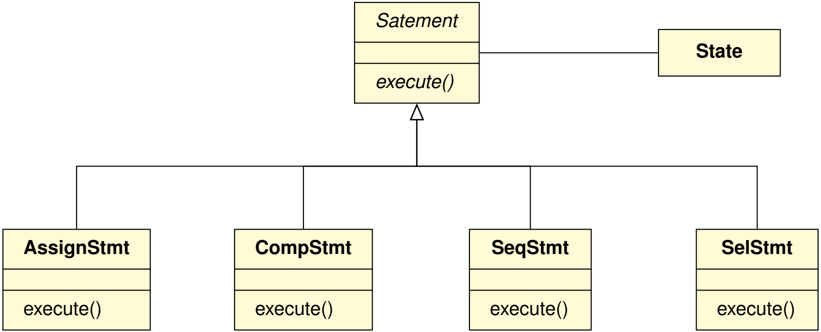



Note the notational difference between the dependence and inheritance relations. Inheritance also implies dependency.

# Example



# Example


Satement
execute()
State
AssignStmt
execute()
CompStmt
execute()
SeqStmt
execute()
SelStmt
execute()

# Plan

# Conclusion

- Models first.
- Apply the right OO Design Principle wherever such a principle is applicable
- Revise your applications from previous laboratories and design their models and identify the places where principles are applicable.