Concepts of Object-Oriented Programming

Instructor: Prof. Peter Müller

Contents

1. Introduction

1.1. Requirements

We have 4 software requirement categories:

- Reuse: quality, interface documentation, adaptability
- Computation as Simulation: real word entities, running simulations.
- GUIs: adaptable standard functionality, concurrency
- Distributed Programming: concurrency, communication

Example: there is no explicit support for extension and adaptability with C, i.e., we must modify the main structure (e.g., Person) definition and methods whenever we want to add a substructure (e.g., Student) → code duplication, difficult to maintain, error prone.

1.2. Core Concepts

1.2.1. Object Model

Object Model: to make the programs reflect the reality they are treating.

- A software system is a set of cooperating objects.
- Objects have state (data), behavior (code) and identity (memory loc).
- Objects exchange messages.

1.2.2. Interfaces and Encapsulation

Interface: publicly accessible fields and methods: describe behavior.
Encapsulation: implementation is hidden behind interface.

1.2.3. Classification and Polymorphism

Classification: hierarchical structure of objects.
Substitution Principle: specialized object is usable if general one is expected.

Polymorphism: the possibility to use a part of a program in different classes.

- Subtype Polymorphism: using the superclass code.
- Parametric Polymorphism: generic class/method that works with different types.
- Ad-hoc Polymorphism: method overloading: several methods with same name and different arguments.

```
bar(Object o){...};
bar(String s){...};
Object x="Hello";
bar(x);
```

→ `bar(Object o)` is used because the static (compile time) type of `x` is `Object`, i.e., the compiler will not know the dynamic type (at runtime).

```
bar(Object a, String b){...};
bar(String a, String b){...};
bar("Hello", "World");
```

→ `bar(String a, String b)` is used because the Java compiler chooses the most specific method.

NB: a method `g` is more general than `f` is any call of `f` could be handled by `g`.

```
bar(Object a, String b){...};
bar(String a, Object b){...};
bar("Hello", "World");
```

→ Ambiguous call. We can solve that by upcasting `a` to `(Object) "Hello"`. In this case, `bar(Object a, String b)` will be used.

Specialization: adding specific properties and/or refining a concept.

1.3. Language Concepts

Dynamic Method Binding: select the method based on the runtime receiver object type and not on the static type. Why? To select the most specific method at runtime.

1.4. Course Organization

1.5. Language Design

What is a Good OO-Language?

A good language should resolve design trade-offs in a way suitable for its application domain.

Design Goals

- Simplicity: syntax and semantics can easily be understood (Pascal, C)
- Expressiveness: can easily express complex processes (C#, Scala, Python)
- (Static) Safety: allow errors to be discovered ideally at compile time (Java, C#, Scala)
- Modularity: modules can be compiled separately (Java, C#, Scala)
- Performance: programs can be executed efficiently (C, C++)
- Productivity: low cost of writing programs (Visual Basic, Python)
- Backwards Compatibility: newer versions work and interface with older versions programs (Java, C)

2. Types and Subtyping

2.1. Types

A type system is a tractable syntactic (based on form not behavior) for proving absence of certain program behaviors by classifying phrases (expressions, methods) according to the kinds (types) of values they compute.

Weak & Strong Type Systems

- Untyped: do not classify values into types (assembly language).
- Weakly-typed: classify into types but do not enforce restrictions (C, C++).
- Strongly-typed: enforce that all operations are applied to the appropriate types (C#, Java, Python).

A type is a set of values sharing the same properties.

- Nominal types: based on type names (C++, Eiffel, Java)
- Structural types: based on the availability of methods and fields (Python)

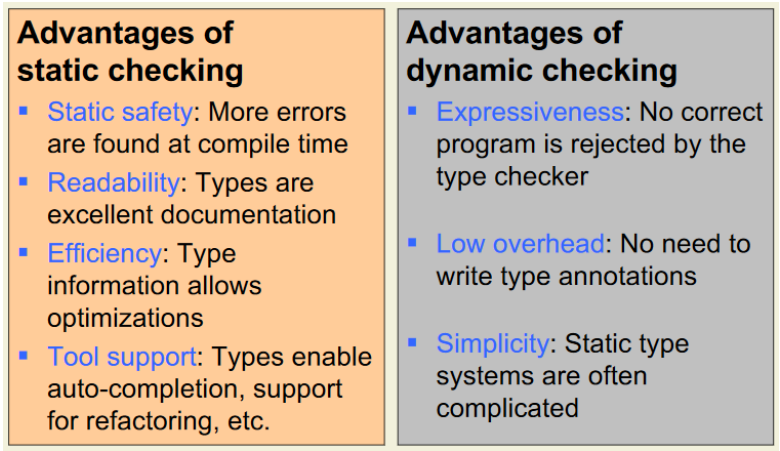
Static Type Checking: each expression has a type + Types are declared explicitly + Type rules are used at compile time.
Dynamic Type Checking: variable, methods and expressions are typically not typed + run-time system checks expected arguments.

Static Type Safety: being able to catch type errors at compile time. This invariant is guaranteed: "In every execution state, the value held by variable v is an element of the declared type of v". This can prevent run-time errors.

Bypass Static Type Checks: C# example:

```
dynamic v = getPythonObject();
dynamic res = v.Foo(5);
```

We can also force Static Typing to Dynamic Type Languages (e.g., mypy for Python): it is called regular typing.



2.2. Subtyping

Syntactic Classification: subtype object can understand at least the messages that supertype objects can understand.
Semantic Classification: subtype objects provide at least the behavior of super objects.

Subtype relation = Subset relation

In Nominal type systems: determine subtype relations based on explicit declarations e.g., `class T extends S {...}`.
In Structural type systems: determine subtype relations based on availability of methods and fields e.g., `class S {m(int){}}` and `class T {m(int){}; n(){}}`.

Subtype objects have wider interfaces than supertype objects:

- Existence of methods and fields → subtypes may add but not remove
- Accessibility of methods and fields → overriding method must not be less accessible (e.g., private, public methods)
- Types of methods and fields
  - → overriding methods must not require more specific parameter types (contravariant parameters rule ≠ invariant rule (e.g., Java): exactly same parameters)
  - → overriding methods must not have a more general result type (covariant results rule)
  - → subtypes must not change the types of fields.

Shortcomings of Nominal Subtyping
→ If we do not have access to the subclasses source code, we can use Adapter (intermediate class) or Generalization. Example:

```
interface Person generalizes Resident, Employee{};
```

→ Nominal subtyping can limit generality (method signature are restrictive). We can introduce interfaces.

2.3. Behavioral Subtyping

What are the properties shared by the values of a type?

Method Behavior

- Preconditions must hold in the state before the method execution.
- Postconditions must hold in the state after the method execution.
- Old-expression is used to refer to pre-states from the postcondition

Object Invariants

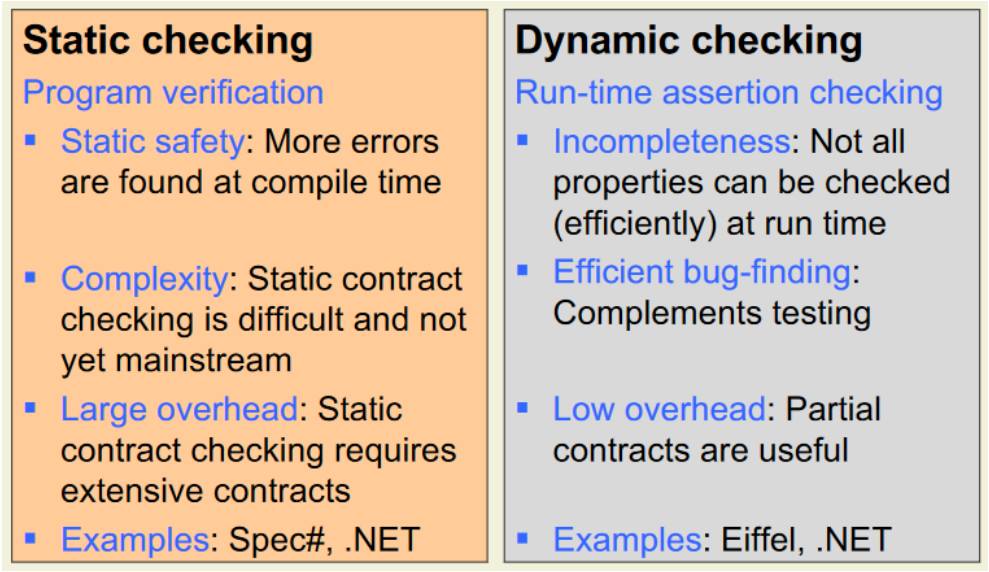
- Invariants describe consistency criteria for object.

Visible States

- Invariants must hold in pre- and post-states (visible states) of method executions but may be violated temporarily in between.

History Constraints

- Describe how objects evolve over time.
- Relate visible states.
- Must be reflexive and transitive.



Contracts and Subtyping: Behavioral Subtyping

Preconditions: overriding methods may have weaker preconditions

Postconditions: overriding methods may have stronger postconditions.

Invariants: subtypes may have stronger invariants.

History Constraints: subtypes may have stronger history constraints.

Static Checking of Behavioral Subtyping

If S.m overrides T.m, check that

- $\text{Pre}_{\text{T.m}} \Rightarrow \text{Pre}_{\text{S.m}}$
- $\text{Pre}_{\text{T.m}} \Rightarrow (\text{Post}_{\text{S.m}} \Rightarrow \text{Post}_{\text{T.m}})$
- $\text{Inv}_{\text{S}} \Rightarrow \text{Inv}_{\text{T}}$
- $\text{Conv}_{\text{S}} \Rightarrow \text{Conv}_{\text{T}}$

Effective Contracts

- $\text{PreEff}_{\text{S.m}} = \text{Pre}_{\text{T.m}} \| \text{Pre}_{\text{T'.m}} \| \ldots$
- $\text{PostEff}_{\text{S.m}} = (\text{old}(\text{Pre}_{\text{S.m}}) \Rightarrow \text{Post}_{\text{S.m}}) \&\& (\text{old}(\text{Pre}_{\text{T.m}}) \Rightarrow \text{Post}_{\text{T.m}}) \&\& \ldots$

Run-Time Checking

Checking all arguments, heaps and results is not possible at run time.

⇒ Define effective contracts to satisfy behavioral subtyping.

Immutable Types

Objects of immutable types do not change their state after construction (no setter).

+ No unexpected modification.
+ No inconsistent states.

Inheritance relation between mutable and immutable types: Do not use optional methods because static safety would be violated (add a setter in the immutable type that throws an exception: Java does that!).

Clean solution: no subtype relation.

3. Inheritance

Inheritance:

- Only one object at run time (Student is a Person).
- Relation fixed at compile time.

Aggregation:

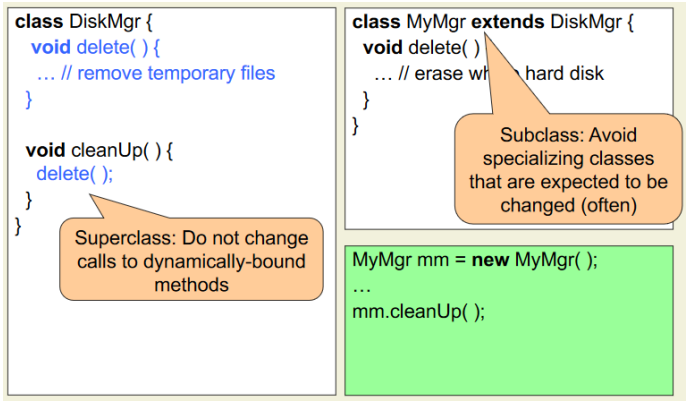
- Establishes a "has-a" relation (Car has an Engine).
- Two objects at run time.

3.1. Inheritance and Subtyping

Subtyping: classification (type aspect).
Inheritance: code reuse (reuse aspect).
Subclassing = Subtyping + Inheritance.

How to reuse code without subtyping (Set/BoundedSet example)?
Solution1: **Aggregation**
BoundedSet uses Set. Method calls are delegated to Set. No subtype relation.
Solution 2: **Create New Objects**
Polygon addVertex(Vertex v) return new Pentagon(...);
Solution 3: **Weak Superclass Contract**
Create an abstract super class for both BoundedSet and Set.
Solution 4: **Inheritance w/o Subtyping**
This solution is supported by some languages.

3.2. Dynamic Method Binding



Rules for Proper Subclassing

- Use subclassing only if there is an "is-a" relation.
- Rely on documentation instead of implementation.
- When involving superclasses, do not mess around with dynamically bound methods.
- Do not specialize superclasses that are expected to change often.

Binary Methods

Methods that take receiver and one explicit argument, e.g., a.equals(b).

- Solution 1: explicit type test (instanceof + downcast).
- Solution 2: double invocation.
- Solution 3: overloading + Dynamic
  (s1 as dynamic).intersect(s2 as dynamic)

3.3. Multiple Inheritance

Ambiguities
Explicit selection (by the client): int w = p.Assistant::workLoad( ); (C++)
Merging methods: return Student::workLoad( ) + Assistant::workLoad( );
Renaming (Eiffel): rename test as takeExam

Diamond of Death
NB: field initialization is done from top to bottom (Person then Student then PhDStudent).
One copy for all virtual inheritance and one copy for every non-virtual inheritance.

Virtual inheritance: the constructor of the top class must be directly called.

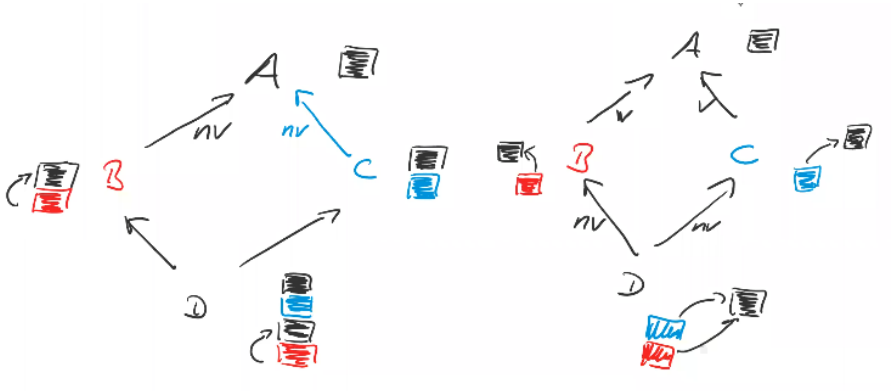
```
class Person {
  Address* address;
  int workdays;
public:
  Person( Address* a, int w ) {
    address = a;
    workdays = w;
  };
};
```

```
class Student : virtual public Person {
public:
  Student( Address* a ) : Person( a, 5 ) { };
};
```

```
class Assistant: virtual public Person {
public:
  Assistant( Address* a ) : Person( a, 6 ) { };
};
```

```
class PhDStudent : public Student, public Assistant {
public:
  PhDStudent( Address* a ) : Person( a, 7 ), Student( a ), Assistant( a ) { };
};
```

Memory Layout: nv → copy fields & v → pointer to the fields



3.4. Linearization

Mixins and Traits: methods and state that can be mixed into various classes.

Traits are made to be mixed in and not instantiated by themselves.

```
trait Backup extends Cell {}
class FancyCell extends Cell with Backup {}
```

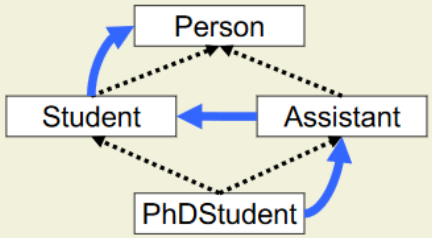
Calculate Linearization

C extends C' with C_1 with ... with C_n

The linearization $L(C)$ is $C, L(C_n) * ... * L(C_1) * L(C')$

Example

```
class PhDStudent
        extends Person
        with Student
        with Assistant
```



L(Person) = Person

L(Student) = Student, Person

L(Assistant) = Assistant, Person

L(PhDStudent) = PhDStudent, L(Assistant) • L(Student) • L(Person)

PhDStudent uses the Assistant's workload.

4. Types

4.1. Bytecode Verification

Programs are compiled to bytecode (platform-independent), organized into classes. Applets get access to system resources only through an API.

Mobile code cannot be trusted: may not be type safe, modify data, expose personal information, degrade performance (DoS). How to guarantee security?

Java Bytecode

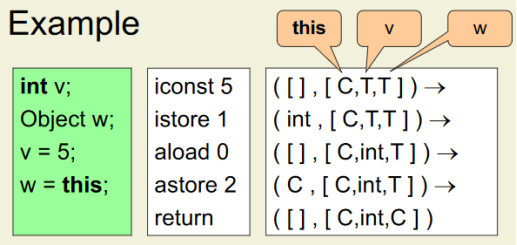
Load, store: to access local variables.

Puts the intermediate result on the stack and pop them whenever needed.

Simulate the execution of the program on the level of types.

How stack and local variable are modified: i:(S,R)→(S',R').

Example: iadd:(int.int.S,R)→(int.S,R).



aload: load reference from local variables to stack.
astore: store a reference to local variable.
iconst: load an int onto the stack.
isore: store an int to local variable.

Instructions can have several predecessors: we must check all combinations.
Consider the smallest common supertype (can ignore interfaces and choose Object type).
Inference: determine the input and output configuration for every instruction.
(+) determine the most general solution that satisfies the rules.
(+) little type information required in class file.
(-) fixpoint calculations may be slow.
(-) solution for interfaces is imprecise and require run-time checks.

Alternative: add type checking to type inference.

Type inference: don't declare types and the compiler can automatically do it.
Dynamic typing: turn off the static type checking.

4.2. Parametric Polymorphism

```
class Queue<T> {
  T elem;
  Queue<T> next;
  void enqueue( T e ) { … }
  T dequeue( ) { … }
}
```
Java

T is a parameter. The client choose the type later.

We can't be sure that T has a specific method. So, we make sure that T is a subtype of a class having that method (upper bound).

class Queue<T extends Comparable<T>>{}

Covariance: if S<:T then C<S><:C<T> → unsafe when variables are written by clients.
Contravariance: if S<:T then C<T><:C<S> → unsafe when variables are read by clients.

Working with Non-Variant Generics (Java)
Solution1: additional type parameters (type parameters for methods)

static <T> void printAll(Collection<T> c)

Solution 2: Wildcards

static void printAll(Collection<?> c)

Different ? could represent different types.
We can use upper and lower bounds for wildcards.
Use-Site Variance:

Random <? extends Person> r= newRandom<Student>();

```
class Wrapper {
  Cell<?> data;
}
```

```
Wrapper w = new Wrapper( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

```
class Wrapper<T> {
  Cell<T> data;
}
```

```
Wrapper<Object> w = new Wrapper<Object>( );
w.data = new Cell<String>( );
w.data = new Cell<Object>( );
```

Compiler

- C<T> is translated to C.
- T is translated to upper bound.
- Casts are added when needed.

In C++, templates are used to allow classes and methods to be parameterized.
Concepts declare syntactical and type constraints.

Template Meta-Programming: calculations could be done in compile time.

Generic Types	Templates
▪ Modular type checking of generic class - Overhead (e.g., upper bounds)	▪ Type checking per instantiation - Flexibility like with structural typing
▪ Run-time support desirable	▪ No need for run-time support
▪ Typically no meta-programming	▪ Meta-programming is Turing-complete

5. Information Hiding and Encapsulation

5.1. Information Hiding

A technique used to reduce dependencies between modules (at compile time).
Class Interfaces: client interface, subclass interface, friend interface.

Express Information Hiding

Java: access modifiers

- Public: client interface
- Protected: subclass + friend interface
- Default access: friend interface (the package)
- Private: implementation (all class methods)

Eiffel: clients clause in feature declarations

- feature { ANY }: client interface
- feature { T }: friend interface for class T
- feature { NONE }: implementation (only “this”-object)

Rules of Overriding

Access Rule: the access modifier of an overriding method must provide at least as much access as the overridden method.

Override rule: a method Sub.m overrides the superclass method Super.m only if the latter is accessible from Sub (private methods are never overwritten).

5.2. Encapsulation

A technique for structuring the state of programs (dynamically) by establishing capsules with clearly defined interfaces.
A capsule can be:

- An object (hidden and exposed parts)
- Object structures
- A class
- All classes of a subtype hierarchy
- A package

It requires a definition of boundaries and interfaces at the boundary.

Consistency of Objects:

- Hide internal representation whenever possible.
- Make consistency criteria explicit (documentation).
- Check interfaces: make sure that criteria are preserved for exported operations (e.g., subclass methods).

How to check interfaces?

- All exported objects preserve the invariants of the receiver (this).
- All constructors establish the invariants of the new object.

Declaring fields private (e.g., in Java) does not give encapsulation on the level of individual object, because one object can change private field of another object of the same class. Eiffel supports object-level encapsulation: feature { NONE }. How to fix for Java?

- All exported methods and constructors of class T preserve the invariants of all objects of T.
- All constructors establish the invariants of the new object.

6. Object Structures and Aliasing

An object structure is set of objects that are connected via references.

6.1. Aliasing

Having different names for the same memory location.

In OOP, an object o is aliased if two or more variables hold references to o

→ efficiency, sharing.

Static (in heap memory): if all references are fields of objects or static fields.

Dynamic (in stack): not static.

6.2. Problems of Aliasing

Aliasing can be used to bypass interface (leaking of reference).

```
public void setElems( int[ ] ia ){ array = ia;}
```

Java deals with this problem by restricting subtyping.

6.3. Readonly Types

Aliases are helpful to share side-effects, and cloning is not efficient.

→ grant read access only (restrict the reference not the object).

- Use supertype interfaces having only getters.
  - No checks that methods are readonly.
  - Clients can use casts to get full access.
- In C++, we can use const Pointers (no field updates).
  - The same cast problem.
  - Const Pointers are not transitive (fields are not const).

Solutions (C++)?

Pure Methods: no field updates, no non-pure methods, no object creation, only overridden by pure methods.

Types

Each class or interface T introduces two types: rw T and ro T.

S<:T⇒rw S <: rw T ∧ ro S <: ro T and rw T <: ro T.

Type Rules: check the "receiver > call" combination.

▶	*rw T*	*ro T*
rw S	*rw T*	*ro T*
ro S	*ro T*	*ro T*

6.4. Ownership Types

Goal? Get control over aliases. The compiler should distinguish internal references from other.

Ownership Model

- Each object has 0 or 1 owner objects.
- Objects with same owner are context.
- Ownership is acyclic.
- The heap is structured into a forest of ownership trees.

At run-time, type information is class and owner of each object:

- Peer: same owner as this.
- Rep: owner is this.
- Any: any owner.

public rep Address addr; different Person objects have different Address objects: no unwanted sharing.

Owner-as-Modifier Discipline = Readonly types + Ownership Types

- Treat any and lost as readonly types.
- Treat self, peer, and rep as readwrite types.

Also:

- Field write e.f=v is valid only if $\tau(e)$ is self, peer, or rep.

- Method call e.m(...) is valid only if $\tau(e)$ is self, peer, or rep, or m is pure.

⇒ rep and any types enable encapsulation of whole object structure, which cannot be violated by subclasses, via casts, etc.

7. Reflection

A program can observe and modify its own structure and behavior.

7.1. Introspection

It is about observing the structure and behavior.

Class Object

```
class Class<T> … {
  static Class<?>     forName( String name ) throws …        {…}
  Method[ ]  getMethods( )                                   {…}
  Method[ ]  getDeclaredMethods( )                           {…}
  Method     getMethod( String name, Class<?>… parTypes )    {…}
  Class<? super T>  getSuperclass( )                         {…}
  boolean    isAssignableFrom( Class<?> cls )                {…}
  T          newInstance( ) throws …                         {…}
  … }
```

Java

Use f.setAccessible(true); before getting the value of a field f.

_ END _