FOOP Summary

Smonman 12021358
September 26, 2024

1 General Goal of Object-Oriented Programming

- Data-encapsulation
- Easy maintainance
- Modelling of the real world
- Easy to extend programs

Object-oriented programming provides abstraction to developers. Large programs cannot be understood fully by a single developer, but abstractions can be. OOP enables us to develop and use these abstractions. Although it comes with a cost.

2 Substitutability

Type S is a subtype of a type T iff each instance of S is usable where an instance of T is expected.

The *Principle of Substitutability* holds if:

- types of input parameters are *contravariant* (weaker)
- types of variables and through-parameters are *invariant*
- types of output parameters are *covariant*

Many things are easier to design, if input parameters would be covariant.

2.1 Behavior

Corresponding methods in S show the same behavior as those in T, if the *Liskov Substitution Principle* holds:

- assertions to be fulfilled by clients (= preconditions) in S are not stronger than those in T
- assertions to be fulfilled by servers (= postconditions) in S are not weaker than those in T
- assertions to be fulfilled by clients and servers (only useful for invariants, history constraints in some cases) equivalent in S and T
- methods in S do not throw more exceptions than those in T (in corresponding situations)

In Java it is not possible to prevent the change of private variables from the outside, since visibility is applied on a class level, not on an object level. In a class, private variables of the same object can be accessed.

Rule three demands, that no variables involved in the invariant are changed to an illegal value. Even though variables could be changed from the outside in Java, it is still possible to fulfill this rule, as you know the restrictions of the variable in the class. Thus, it can be satisfied to only have the allowed values in a variable. This is really difficult to enforce if the visibility is set to protected not private.

3 Assertions

3.1 Theory

In theory there are multiple ways to express assertions.

- Logics
- Algebra

- Programming language

Relationships between assertions of subtype and supertype are checkable. Formally this might be harder, but it is easy to extend an assertion either with an `AND` ($\wedge$) or and `OR` ($\vee$).

3.2 Practice

But in practice, assertions are not expressible and relationships cannot be checked because:

- Programming languages have insufficient support (informal comments, usually ambiguous)
- Assertion checking is only possible at runtime, checking if the program is well defined is only possible at compile time
- Checking assertions at runtime has a runtime penalty
- Checking assertions at runtime, which has functions in it, potentially leads to an infinite loop. Therefore, assertion checking is switched off, when checking assertions.
- Clients have not enough information about the object state. Clients need to know if the preconditions are fulfulled prior to calling the method, but they cannot know the full object state because of visibility restrictions.
- Therefore, data hiding is in direct conflict with Design-By-Contract
- Object state changes in non-predictable way. This might happen due to
  - Concurrency, or
  - Aliasing

  That means between the time of checking whether the preconditions are true, and the client actually invoking the method, the state of the server could have changed.

```java
class IntSet {
    /*
     * true iff x is in set
     */
    public boolean find(int x) { ... }

    /*
     * immediately after insertion x is in the set
     */
    public void insert(int x) { ... }
}
```

```java
IntSet set = new IntSet();
set.insert(1);
boolean a = set.find(1);
do_something_not_using_set();
boolean b = set.find(1);
```

It is not given, that `a` and `b` are both true. The statement "immediately after" is not well defined.

A concurrent thread (which could have been spawned by the constructor of the set) could have deleted the 1. Or, the constructor could have introduced an *alias* that causes the invocation of `do_something_not_using_set` to alter the set. All this does not conflict with the assertions.

Also the assertions do not mention that `find` does not remove the element afterwards. It is not practical or possible to list everything, that a method does *not* do. Thus, often, it is implicitly assumed, that each method does nothing, except the things mentioned in the pre- and postconditions.

3.2.1 How to Solve the Problems

- *Invoke method to check if the element is in the set*

  This is often a bad solution because there is nothing useful to do if the check fails. Also, `find` could have side-effects.
- *Prevent aliasing alltogether*

  This is very expensive to ensure and extremely restrictive. The code becomes linear.
- *In case of concurrency, always ensure atomic actions*

  This is often a good idea, but can be expensive. There is no well defined model for synchronisation if every action is atomic, therefore handling deadlocks and lifelocks becomes hardly manageable.
- *Avoid unexpected side-effects (e.g., in the constructor)*

  This is useful. Since this is not easily formally expressed, it is the task of the developer to ensure this. Developers should be aware of this. This also entails, that one has to trust other developers *in the same team* to be aware of this. Do not trust other developers. Defensive programming is only useful at specific points in the program, not everywhere.

3.3 History Constraints

History constraints are assertions regarding time or change. For example, if the content of a variable may only increase, this is expressible via a history constraint, not an invariant.

The two typical use cases are:

- defining the change of a variable or the content over time
- defining sequential invokation: method `x` is only callable directly after method `y`.

The first rule is *server responsible* while the second one is *client responsible.* Server responsible history constraints resemble invariants, thus they can be more restrictive in subtypes. Client responsible history constraints allow more invocation sequences in subtypes.

3.4 Suggestions

- It is more important to think about whether the client or the server is responsible for a specific assertion than to figure out the type of assertion.
- Do not try to consider all possibilities. This is too expensive and often not possible.
- Be predictable; “No hidden anything”. Rely on the usual behavior of known methods.
- Use design rules
- Use all available information in assertions. This includes history constraints.
- Avoid unnecessary dependences!
  - no avoidable or unnecessary assertions
  - no unneeded invocations or parameters
  - no unnecessary visibility of variables and methods
  - only well-considered parameter types
  - code only at the lowest level in class hierarchy (= avoid inheritance, use subtyping)

4 Significance of Names

Names are abstractions of the object behavior, method behavior and variable properties. Names resemble informal assertions.

- Names support *intuition.* This does not mean, that comments are not needed. Comments are used to provide further details. E.g. for a method called `insert`, it should provide information on how something is inserted, when something may be inserted and how it could fail. Additionally, comments are fundamental to abstractions.
- Intuitive names cause programmers to be *predictable*

- Names of types provide *semantic* information. Not only for the developer, but also for the compiler.

4.1 Example of Structural Types

Two *structural* (= anonymous) types in a subtyping relation:

```
{ String name; String address(); }
{ String name; String address(); int regNr }
```

A compiler can only check subtyping relations on a structural level. For *nominal* types, the developer has to specifiy the subtyping relations. A subtyping relationship is defined by the availablility of members. In theory we mainly use structural types, in practice, we mainly use nominal types.

4.2 Difference between Nominal and Structural Types

	structural	**nominal**
type equivalence	same structure	same name
specified at	one or more places	exactly one place
subtyping	implicit	explicit
usability	simple	more difficult
plug & play	easier	not that easy
name conflicts	possible	easily possible
accidental relations	possible	unlikely
readability	not so good	better
behavior abstraction	no	yes

For instance, in C every type structual, except `struct`.

Notably, comments or *informal* specifications and remarks are only possible with nominal types, since structural types might be defined multiple times.

4.3 Nominal Types ensure Properties

```
class SortedList<A extends Comparable<A>> { ... }
```

The property *sorted* is hardly directly checkable, except by class membership. Just because another class extends this class, we can assume that it is also sorted.

Each instance of `SortedList<T>` is initialized by a constructor in `SortedList` and its subclasses. That means, upon instance creation, the property that the list is sortes is given. It is not possible to create an unsorted list, and somehow fake it into the hierarchy of `SortedList`. Therfore, if the constructor creates a sorted instance, and all other operations do not disturb this, it holds that the list is always sorted.

4.4 Structural Types have Benefits

- Structural types are great bounds for bounded genericity.

  For instance, in the aforementioned example, only subtypes of `Comparable` are allowed for the type substitute `A` in `SortedList`. This does not mean the method has to be implemented.
- When using plug & play on software components
- When introducing supertypes of already existing subtypes

4.5 Genericity with Anonymous Bounds

Same example in Java and Ada.

```
class SortedList<A extends Comparable<A>> { ... }
```

```
generic
    type T is private;
    with function compare(x, y: T) returns Boolean
package SortedList
```

In Java, `A` has to inherit from `Comparable` only because it provides the method `compareTo`. Ada is an example with anonymous bounds, where the type `T` only requires to have a function `compare` with two parameters of the same type and a Boolean return value. The definitions in Ada can be thought of as two generic parameters, a type and a function. If the function has the same signature (ignoring the name), it can be used.

4.6 Kinds of Bounded Genericity

Genericity is needed since there is one thing not possible with subtyping. It is not possible to have covariant input parameters, and as a result of that it is not possible to have binary methods. Binary methods are methods with the same type in multiple input parameters. For example, in order to compare two objects, two input parameters of the same type are needed.

It is not possible since binary methods need covariant input parameters but only contravariant input parameters are allowed.

- *Genericity based on System $F_{\leq}$* (F-bound)
  $$S \leq T\langle S\rangle$$
  This technique can deal with binary methods, but only for one inheritance level. Here, it is possible to have recursive type parameters. For example: `Integer` $\leq$ `Comparable<Integer>`. With this recursion, binary methods are possible.

  The problem with this technique is, that this recursion is only possible at one level in the hierarchy. It is not possible to have a subtype of Integer with a binary compare method, only with a compare method to Integer in this case.

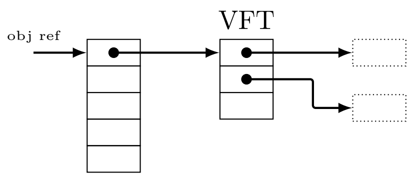
  This is used e.g. in Java.
- *Higher Order Subtyping*
  $$S <\# T \text{ if } \forall U : S\langle U\rangle \leq T\langle U\rangle$$
  In Higher Order Subtyping, both S and T are generic parameters. It defines, that if S uses a generic type U, then it also is a subtype of T when it has the same generic type U. For example: `LinkedList<Integer` ↪ `>` $<\#$ `List<Integer>`.

  The problem with Higher Order Subtyping is, that it does not imply subtyping. It provides binary methods directly, but no substitutability. It cannot be used as a base for OOP, only as a base for genericity.

  This is used e.g. in C++.

5 Dynamic Dispatch

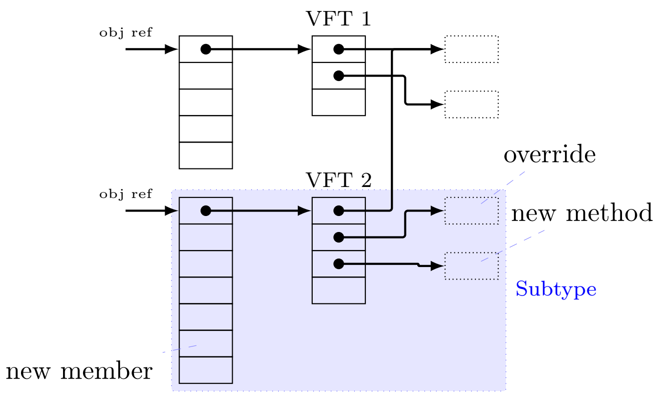
Dynamic dispatch is the process of finding the correct implementation of a specific method on a specific object. The Virtual Function Table (VTF) is the *dynamic type* of the object.



For compiled languages, each method is referred to via a number. The number is then used to find the offset in the VFT. Variables are offsets from the object reference.

5.1 Single Inheritance

When inheriting, the VTF of the superclass is simply copied. Function overrides are implemented by changing the pointer in the new VFT. Additional methods are easily appended to the new VFT, likewise new members of the object are also appended.

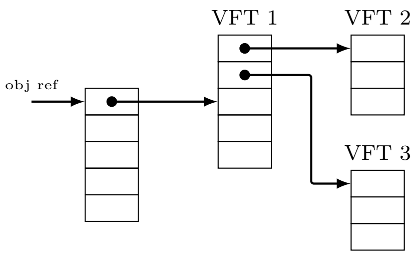


5.2 Dynamic Dispatch with Interfaces

For multiple inheritance, we need multiple VFTs. Negative indices are used to reference other VTFs in the class VFT.

Further VFTs are located in the class VFT and not the object itself, as this minimizes memory consumption. Furthermore, there is no speed trade off, as the same operations are already undertaken to use the original VTF.

The *declared* type of the object is used in order to determine which VFT should be used.



The tradeoff of this mechanic is, that further VFTs that represent e.g. interfaces have to be created per class that implements them, not per interface. Another one would be, that only a single inheritance does not change the underlying structure. As soon as there are two interhitance relations, the structure has to be altered to support this change.

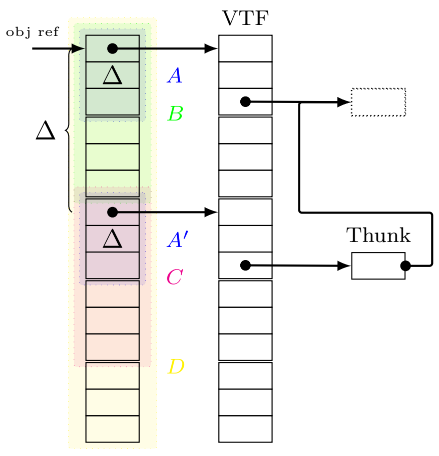
In Java, for example, every class is a subclass of `Object`. That means, any implementation or inheritance requires multiple VFTs.

5.3 Multiple Inheritance

In contrast to dynamic dispatch with interfaces, multiple inheritance also needs to figure out how to work with multiple object variables, not just inherited methods. C++, for instance, allows multiple inheritance.

$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd}$$

As easily seen, to cast e.g. D to C, the object reference is moved to the start of C. Analogously, an up-cast from C to e.g. A can be performed. The required offset Δ is stored in *either* the object, or the VFT, not in both places, usually with an offset of 1.



Before comparing object references, the delta must be subtracted.

5.3.1 Method Invocation and Thunks

Method invocation with multiple inheritance in general:

```
1 load [objectReg + #VftOffset], tableReg
2 load [tableReg + #deltaOffset], deltaReg
3 load [tableReg + #selectorOffset], methodReg
4 add objectReg, deltaReg, objectReg
5 call methodReg
```

The `#VftOffset` is the offset from the start of an object until the pointer to the VFT. Similarly, the `#deltaOffset` is the offset from the start of an object or VFT until the delta. The `#selectorOffset` is the offset to the method that should be invoked.

In line 1, the location of the VFT is calculated. Next, in line 2, the delta value is fetched. Subsequently, in line 3, the address of the invoked method is computed. Line 4 calculates the correct object referece, based on the used VFT. This is important, as otherwise the variables in the invoked method might not be referenced correctly.

Since this implementation is really expensive, in reality, some optimizations have been made. One observation was, that $\Delta = 0$ in most cases. Lines 2 and 4 are generaly skipped, if $\Delta = 0$, otherwise, a so called *thunk* is used as a proxy method before calling the actual method. The thunk does nothing else than adjusting the object reference prior to calling the requested method.

```
add objectReg, #delta, objectReg
jump #method
```

5.4 Dynamic Dispatch in a Dynamic Language

In Python e.g. the definition of a function can change at runtime. In contrast to compiled languages, the compiler cannot build the previously shown structure of VFTs.

In dynamic languages upon invoking a method, the method name is searched in the object. If it could not be found, the super type is searched and so on.

5.5 Optimizations

- *Dynamic Caching*

  This is usually used in dynamic languages. When in the same location of the program simply jump to the method invoked last time. Only afterwards, compare the classes to check if it is the correct one.

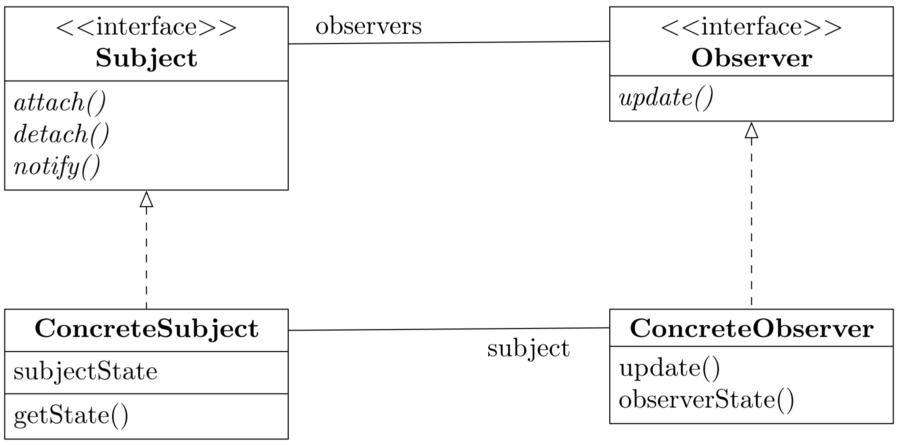
- *Method Inlining*

Method inlining means the method will not be called, rather the instructions of the method will be copied to the location where it would have been called. This can only be done, if we know for certain which method to call. It does not work with dynamic dispatch, it requires static dispatch. Method inlining and other optimization possibilities that come with it make static dispatch much more performant than dynamic dispatch.

6 Design Patterns

6.1 Observer Pattern

The observer pattern defines a 1–to–n relation between objects. There are n objects observing one object, which keeps the dependent object informed about state changes.



6.1.1 Use Cases

- An abstraction with two aspects, where one depents on the other
- A state change causes further state changes without static knowledge of the objects to be changed
- Objects shall be informed about something but these objects are not satically known (loose coupling)

6.1.2 Consequences

- The subject and observer are only coupled abstractly. The never comunicate with a concrete implementation. Therefore, it is possible to have coupling between different layers, e.g. in a layered architecture.
- Broadcasts happen automatically. Observers can be added and removed at any time. It is not necessary to explicitily get state information from another object.
- Unexpected updates are possible. The subject could change without notification to the observers if, for example, an error occurs.
- Updates or too many updates could be expensive.

6.1.3 Implementation of Observer

Usually the subject is passed to the `update` method as an argument. This helps with differentiation.

There are two ways how a notify could be triggered:

- A client of a subject could trigger it. For example, when pressing save.
- The subject itself could trigger the notify if something changes in the object.

The first idea often produces not enough notifications, and is usually error prone, while the second one produces too many and often unnecessary updates.

Additionally, it is important that the state of the subject is consistent prior to a notification, and that observers dereference them prior to deletion.

There are also two ways how data can be transmitted upon receiving an update:

- *Push Model*

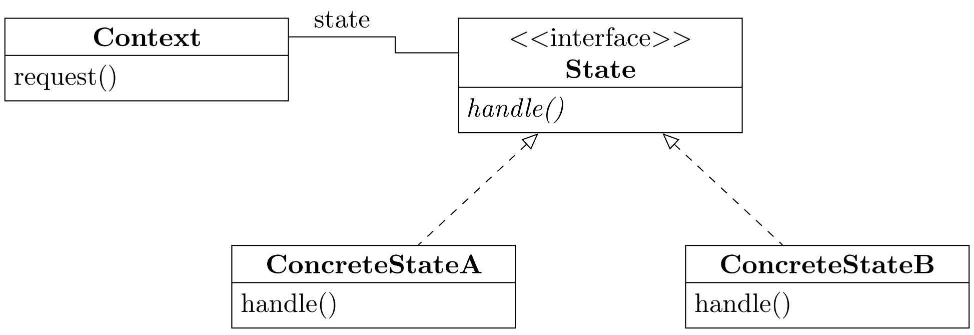
  Pass data as arguments to the `update` method.
- *Pull Model*

  Send minimal information via arguments. Observers have to seperately request data.

6.2 State Design Pattern

This allows an object to change its behavior if the internal state changes. We want to avoid explicit state checking and branching.

By switching the contents of the state reference, the behavior can be changed. Internally, the explicit branching is replaced by dynamic dispatch.



6.2.1 Consequences

- localizes state-specific behavior
- separates behavior in different states
- easy extensible with states and state transitions
- code distributed over many classes (disadvantage)
- state transitions are explicit, they cannot be changed in background
- all states are self-consistent because of atomic state transitions, as long as assignment is atomic
- state objects can be shared by several contexts, as long as the data is bound to the context

6.2.2 Implementation

Who causes or defines when a state transition happens?

- *Context*

  Subsequent state does not depend on the current state.
- *State*

  Strong dependence between subclasses. For example a traffic light.

But it is also possible to use a *state transition table* in the context, even if states depend on each other. Using this table, the context has information about these dependencies. This focuses on state transitions not on behavior. Also, this transition table could be generated, as it is done by parsers and lexers. As soon as there is a transition table, the states can be thought of as contained in a statemachine, where the table describes the language.

The idea would be to have a grammer describe the language, which is then used to generate the state transition table. Explicitly writing the table is error prone.

States often implement the *singleton* design pattern.

In languages with dynamic inheritance (e.g. Self, Smalltalk), this inheritance could be used to express states.

7 Inheritance vs. Delegation

```java
public class A {
    public String foo1() { return fooX(); }
    public String foo2() { return fooY(); }
    public String fooX() { return "foo1A"; }
    public String fooY() { return "foo2A"; }
}

public class B extends A {
    A delegate = new A();
    public String foo1() { return delegate.foo1(); }
    public String fooX() { return "foo1B"; }
    public String fooY() { return "foo2B"; }
}
```

The difference here can be shown by an example:

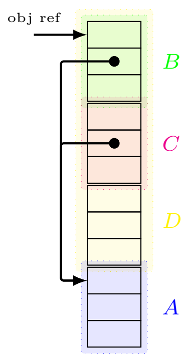
```java
A a = new A();
B b = new B();

b.foo1() // => foo1A
b.foo2() // => foo2B
```

The result for the call to `foo2` is reached because of dynamic dispatch.

7.1 Virtual Inheritance (C++)

When using virtual inheritance in the same example as mentioned in Section 5.3, D does not have two implementations of A. Virtual inheritance can be thought of as delegation rather than inheritance. Both B and C delegate to the same A. This also means, that you need an *object* of A to refer to it.

Since delegation and inheritance have slight differences, for virtual inheritance this delegation has to be tweaked to look like inheritance.



The data for the object A is not stored twice, once in B and once in C, but rather appended, and only referenced. In order to have inheritance like method invocation, a delta to A is used. This delta is then added prior to a method call, and subtracted afterwards.

7.2 Composition

Composition is an overarching term, where both inheritance and delegation fall into. But, inheritance and especially multiple inheritance are error prone and should be avoided. The preferred way for composition is delegation.

The advantages of delegation are the clear semantics and readability. It is easy to see which methods are actually invoked.

7.3 Inheritance Anomaly

Inheritance does work nicely together with concurrent programs, because of synchronisation.

Also, adding methods via inheritance might destroy implicit and explicit assumptions of other methods. The usual example is a set, that does not have a delete method. When using this set, the client can be sure, that after adding an element, this element is surely in the set, as there is simply no functionality to remove one. Now, when creating an subtype of this set, with a delete method, assumptions about the set might no longer hold, since in the end, it is not known, which concrete type is used at runtime. Today, it is accepted, that such behavior has to be permitted via informal comments, and assertions; e.g. if one postcondition states, that the element is in the set after the insert, it is not allowed to extend the set with a delete method.

These problems do not arise with delegation.

8 Parallelism vs. Concurrency

The difference between parallelism and concurrency lies in the goals they try to achieve. Parallelism has the goal of performance, where something is done in parallel to finish quicker. Concurrency, on the other hand, simply has the goal to have multiple streams of execution running.

In parallel programming a goal is to avoid synchronisation. For example by only having one thread be able to modify data.

For instance, a webserver that handles each request in another thread would be using concurrency, not parallelism.

9 High Performance Computing in OOP

Experience shows, that object oriented programs are usually around 40 times slower than procedural programs. Even though dynamic dispatch is slower than static dispatch, it does not explain the stark difference in speed. Actually, no object oriented mechanism does, rather the development process is responsible.

Performance only comes when designing with performance in mind. Usually, object oriented projects do not have performance as their number one goal.

Programs that do not use inheritance are faster. But not because of the lack of inheritance, rather because of the lack of "convenience code".

Garbage collection also does not have all too much impact on the performance. Rather it is important to only use code that is strictly necessary. Don't rely on existing code, just because it is convenient.

9.1 Parallel Programming in OOP

Object oriented programming usually provides great tools for concurrency:

- *Actors*

  An actor consists of a single thread and a set of private variables. Variables cannot be read or modified by another actor. Actors can only interact with other actors via messages. The only synchronisation has to happen when sending a message to an inbox queue of another actor. In this model, shared memory is not possible.
- *Active Objects*

  An active object is like an object running in a separate thread. This idea was firstly brought up by Smalltalk.
- *Concurrent Threads*
- *Monitor Concept*
- *Map-Reduce* and similar applicative frameworks

But not so much for parallel programming. A key factor of parallel programming is the independence of data blocks. The data layout must follow what the parallel algorithm requires. This is in direct conflict to object oriented programming.

Object oriented data layouts do not allow for efficient caches. For example, when having multiple objects, which should be searched by name, there are many pieces of information in between the names. This makes iterating and caching more inefficient.

In parallel programming data is more important than the actions.

Also, parallel programming is usually used on a process level, not a thread level. To increase performance, trying to use processes in parallel is key.

10 Automatic Testing and Verification in OOP

In order to have automatic testing, first a clear definition of the expected behavior of the system is required. For procedural and functional programs, this is easy. The semantics of the program can be defined using for example λ-abstraction. But for object oriented programs this is not possible, since semantics are defined in informal comments. Only procedural parts of the program can be defined.

Automatic testing and verification is possible for procedural and functional programs if the semantics can be clearly defined. For object oriented programs on the other hand, this is not possible. Only explicit test cases can be used.

Some specific aspects can still be verified automatically. For example, synchronisation and deadlock-freeness by using model checking. But these approaches are only feasable for small programs.

11 Functional Object Oriented Programming

- *Point of View*

  Many think object oriented is just organisational paradigm and the logic can be written in another paradigm
- *Contradiction*

  A object always has a state, with behavior and identity. In functional programming there is *referential transparency*, which means no state, no side effect and no identity.
- *Practice*

  No referential transparency for input and output. Inner parts of functional programs are referentially transparent.
- *State of the Art*

  Separate functional part from the object oriented part. Procedures may call functions, but functions must not call procedures.

  If the data is kept separate, parallelism can be added.

12 Type Inference

Type inference can be done with a rather simple algorithm:

- use a fresh type variable where no type is known
- use *type unification* where types should be equal (binding type variables)

But this does not work for object oriented programs, especially when using subtyping. It is used for genericity, but as already established, that is not subtyping. Type unification is not wanted in object oriented programs, as we might want to supply a subtype.

Semi-unification can be used instead of unification, only for functional programming or genericity. For supportung subtyping this problem is undecideable.

13 Lambda Expression

A lambda expression is not a function but a *closure*, because variables in its environment are also accessable.

Lambda expressions allow type variables. The compiler can infer the type in such a case, since a lambda expression cannot extended with a subtype. And when no subtyping occurs, type inference is possible.

13.1 Applicative Programming

In functional programming control structures can be implemented using higher order functions. Applicative programming is the extensive use of complex predefined control structures to write programs. These control structures can become very specific, but are in turn very expressive. It is easy to quickly program large project only using given control structures.

A disadvantage of applicative programming is its bad readability.

Applicative programming in Java can be done with Java 8 Streams.

13.2 Lambda Expressions in Java

Any object can be a lambda expression as long as the name of the method is known. This may be defined by style guidelines. As syntactic sugar, the `@FunctionalInterface` annotation only allows a single method in an interface, voiding the need to know the functions name.

```java
@FunctionalInterface
interface X { void anyname(String s); }

@FunctionalInterface
interface Y { int op(int a, int b); }

X x1 = (String p) -> System.out.println(p); // type is optional
Y x2 = p -> {
    System.out.println(p);
    System.out.println(p);
};

Y y1 = (l, r) -> l + r;
Y y2 = (l, r) -> {
    return l * r;
};

x1.anyname("Hello World!");
x2.anyname(y1.op(1, 2) + " " + y2.op(3, 4));
```

In Java, some side effects are forbidden when using lambda expressions. Variables can only be used as if they where declared `final`. This is done to avoid implicit side effects, which are hard to find.

Even if the annotation `@FunctionalInterface` is just visual, it does signal that the method is used in a functional environement. That means, design by contract and therefore assertions are not necessary.

14 Structural Programming

Programming can be reduced to just three control structures:

- loops
- branches
- sequential flow

Structural programming is extremely simple. Still, today we need higher abstraction, thats what object oriented programming provides.

15 Simple Programming

The original JavaScript, Lisp or even the lambda calculus are extremely simple. But this also brings disadvantages. It is hard to program in lambda calculus or Lisp, and in JavaScript there are not enough restrictions. Restrictions, like a static type system help the developer.

A perfect programming paradigm combines all the wanted aspects, like simple language vs. simple programming, abstractions, different programming styles, different paradigms, dynamic vs. static together; but, not everything everywhere. Just where it is needed.

That means, a large project should be divided into smaller parts, where each part should use the lanugage and paradigm best suited for the task.

16 Dynamic Languages vs. Static Languages

Static languages are often used by knowledgeable programmers who want as much work done by the compiler as possible.

Dynamic languages are often used by experts in a domain other than informatics or programming. For example in data science or physics, Python is used often.

That means, both dynamic and static languages are important and have a right to exist. Combining them is hard. But it is easier to start with a statically typed language, where types are removed afterwards, than starting with a dynamic lanugae like JavaScript and enforcing types afterwards (TypeScript).

The combination of dynamic and static is only possible in a useful way in object oriented languages, since dynamic dispatch is required with object oriented programming.

16.1 Glue Languages

Glue languages like Python or PHP are designed with the goal of combining already existing applications into a single unit. Since external applications cannot be assigned a type, and programs written in glue languages are often rather small, static typing is unnecessary or not possible at all.