# Composed Methods:

→ out of calls to other methods
AND
→ all have the same level of abstraction
• mixture of abstraction levels is a poor composed method

→ should contain one brainful of detail ← method length.
↓
closely related details together
• separate it from irrelevant details
→ main flow, separate from exceptional flow.

```
void compute() {
  input();
  flags|= 0x0080;
  output();
}
```

• lack of abstraction level
• direct access only in operation methods of "OSP-Model"
• direct access → lack of intention.

Code is easy to understand, when it flows. Abruptly shifting AL breaks that flow.

• logic - data - detail : together : method/object/package

Composed Method based on facts not speculation.

• unit test → all details in method (in front of me) than it's easy to compose methods sensibly.

# Intention-Revealing Names:

→ name methods after what they are intended to do.

communicate intent in the name
↓
restraint is required
not blur out everything about the program.
• to see the implementation detail look at the method body.

communicate other information in other way (implementation strategy)

• communicate distinction from the invokers perspective.

```
Customer.linearCustomerSearch(String id)
```

This might seem superior to:

```
Customer.find(String id)
```

linear search → `Customer.find(String id)`

hash table search → `Customer.fastFind(String id)`

# Method Visibility:

→ make methods private as possible

→ visibility reveals the intention.

→ more methods revealed to outside → less flexibility

→ reveal as little as possible.

use narrow interface → too-narrow interface costs clients more.

[Public: semantische Invariante

- responsibility to maintain it → every change breaks the callers
- useful outside of the package

Protected:

- useful for subclasses

static

→ only replacement for constructors.

Pattern:

→ use restrictive visibility at the beginning.

→ reveal method later

Flow: that can happen:

private methods $\xrightarrow{\text{get}}$ public $\xrightarrow{\text{put to}}$ interface.

# Method Objects:

→ turn complex methods into their own objects.

1. Create a class named after the method. For example, `complexCalculation()` becomes `ComplexCalculator`.
2. Create a field in the new class for each parameter, local variable, and field used in the method. Give these fields the same names as they have in the original method (you can fix names later).
3. Create a constructor that takes as parameters the method parameters of the original method and the fields of the original object used by the method.
4. Copy the method into a new method, `calculate()`, in the new class. The parameters, locals, and fields used in the old method become field references in the new object.
5. Replace the body of the original method with code that creates an instance of the new class and invokes `calculate()`. For example:

```
complexCalculation() {
    new ComplexCalculator().calculate();
}
```

6. If fields were set in the original method, set them after `calculate()` returns:

```
complexCalculation() {
    ComplexCalculator calculator= new ComplexCalculator();
    calculator.calculate();
    mean= calculator.mean;
    variance= calculator.variance;
}
```

→ data & logic together

→ single responsibility principle

# Overridden Method:

→ override methods to express specialization.

• Well-composed methods in the superclass give you enabling points for many variations.

→ template design pattern.

→ invoke superclass method only in the same method of subclass.

→ large superclass ⇒ don't copy code to subclass
• someone change superclass and it will break ⇒ Programming by coincidence.

# Overloaded methods:

→ provide alternative interfaces to the same computation.

→ flexibility by passing different types of parameters.
↳ invoker doesn't need to make conversions.

→ don't use for overloaded methods different types of return parameters.
→ give different computations different names.

# Method Return Types:

Function have a return parameter !
Procedures doesn't have any.

- pick the most abstract return type
  ↳ broadly applicable methods.
  → flexibility: no need to change in the future
  ↳ hides the implementation detail
  → Collection instead of List

→ return interfaces instead of o

# Method Comment:

→ Comment methods to communicate information not easly read from the code.

→ many comments are completly redundant.

# Helper method:

→ create small, private methods to express the main computation more succintly. → SRP → helper methods

→ hide temporarily irrelevant details
↳ express your intention through method name

# Conversion Methods/Constructors:

→ express conversions between objects

Method converts the source object into destination.

```
class Polar {
  Cartesian asCartesian() {
    ...
  }
}
```

→ need to modify source object
→ unbounded number of potential conversions
↳ it becomes unwiedly.

Constructor conversion:
→ takes the source object as parameter and returns a destination object.

For example, File supports a conversion constructor that converts a String representing the name of a file into an object suitable for reading, writing, and deleting. While it would be convenient to have String.asFile(), there is no end to the number of such conversions so it's better to have File(String name) and URL(String spec) and StringReadStream(String contents). Otherwise, String would have an unbounded number of conversion methods.

unbounded number of conversion for String.

# Complete Constructor:

→ write constructors, that return fully formed objects.
→ overloaded constructors → all should refer fully formed object.

```
new Rectangle(0, 0, 50, 200);
```

→ all 4 arguments are required
→ good communication to readers, about prerequisites

```
Rectangle box= new Rectangle();
box.setLeft(0);
box.setWidth(50);
box.setHeight(200);
box.setTop(0);
```

→ which parameters are required?
→ no way to figure it out.

# Factory Method:

- express more complex creation as a static method on a class rather than a constructor.

→ return more abstract types than a simple constructor
↳ subclass or interface.

# Internal Factory:

```
getX() {
  if (x = null)
    x= ...;
  return x;
}
```

This is plenty for
complicated, it can p

- common in lazy initialization

```
getX() {
  if (x = null)
    x= computeX();
  return x;
}
```

- calculation is complex, than delegate it to a helper method.

# Collection Accessor Methods:

```
List<Book> getBooks() {
  return books;
}
```

→ maximum flexibility for the clients | make any operation on collection.

but

→ creates a variety of problems
↳ change collection invariants
↳ content can be invalidated

all-purpose access

```
List<Book> getBooks() {
  return Collections.unmodifiableList(books);
}
```

Instead, offer methods that provide limited, information in the collection.

```
void addBook(Book arrival) {
  books.add(arrival);
}
int bookCount() {
  return books.size();
}
```

If clients need to iterate over the elements of th
that returns an iterator:

```
Iterator getBooks() {
  return books.iterator();
}
```

→ provide limited, meaningful access to the information in the collection.

→ clients want iterate over the collection, then provide the iterator.