state:

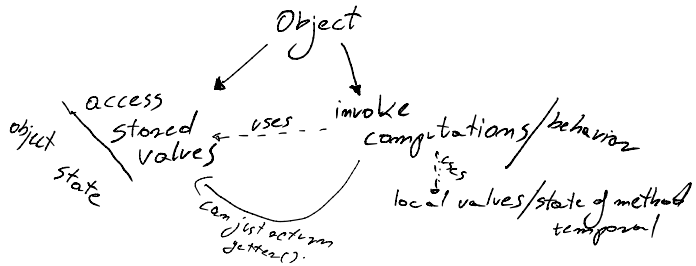
"compute with values, that changes over time"

- managing state
  → put similar state together
- OO-style everything has a state
- Value-style → no state

two pieces of state are used together
two pieces of state have the same lifetime.
have them close to each other.
→ local consequences
→ symmetry

Object: behavior, that operates on state
control of flow → logic of code

Access:



Decision: What to store?
What to compute? / store-versus-compute decision!

Direct Access:

downside: reflects to implementation detail

between objects
→ information hidding

within an object / direct access to state within a class.
→ JOSE/Same level of abstraction
→ in Constructor/

```
doorRegister= 1;

with:

openDoor();

or, with objects:

door.open();
```

direct access / clarity, directness, but no flexibility

indirect access / flexibility

rules: direct access in
- constructor
- accessors methods

→ insight package
→ insight a single class
→ all its subclasses

Indirect Access:

- indirect access for clients to the state.

! most accesses to state outside the class ⇒ design problem
→ low of demeter
→ Tell don't ask

! impl. calculation ⇒ uses thousands of getters to get values for the computation.
↳ where it belongs to ?
separation of concerns.

use indirect access
↳ for coupled data | logic & data together
rate of changes

coupling very direct

loose coupling

```
Rectangle void setWidth(int width) {
  this.width= width;
  area= width * height;
}
```

breaks DRY on data

```
Widget void setBorder(int width) {
  this.width= width;
  notifyListeners();
}
```
← indirect access

best option indirect accessor method
→ listener must be there.

Common state :

- operation shares the same data elements (they have also state)

example: cartesianPoint
↳ always two data x and y.

data elements —have→ values
stored as
fields elements

operations need this data elements to make computation.

→ objects of the same class have always the same data elements.

```
class Point {
  int x;
  int y;
}
```

data elements = field elements
different values

- direct coupling data elements.
- doesn't depend on other data (fields).
- data together

clear to read
not flexible = justified by the nature of the object.

Variable state:

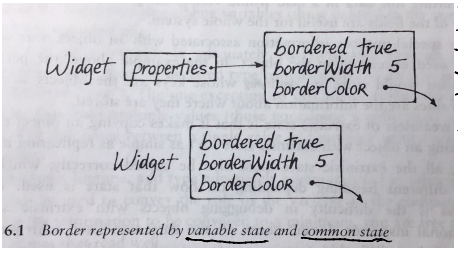
- objects of the same class have different data elements.
- use for fields in object, that may or may not be needed depending on usage.

```
class FlexibleObject {
  Map<String, Object> properties= new HashMap<String, Object>();
  Object getProperty(String key) {
    return properties.get(key);
  }
  void setProperty(String key, Object value) {
    properties.set(key, value);
  }
}
```

← the same object has different data elements depending on how it is used.

← one data element need some other data elements

variable vs. common state



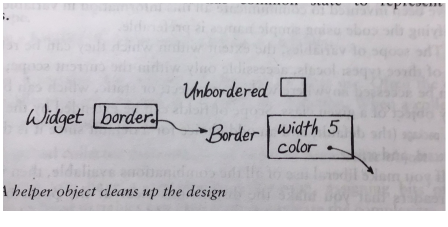
6.1 Border represented by variable state and common state

variable state
→ data element needs other data
→ object of the same class can have other data elements
→ indirect access for data elements
→

common state
→ direct access
→ direct coupling
→ data together
→ rate of change / state changing

• violates the lifetime principle : all variables should have the same lifetime.

↙ helper object.



A helper object cleans up the design

variable state : Widget

common state : Border

local Variable:

= use for short scopes

- common roles
  - → Collector: collects information for the later use.
  - → Count: collects the count of some other objects.
  - → Explaining: reducing complexity / smell code destroy long, complicated expressions.
- common uses
  - → reuse: need to use the same value over and over.
  - → element: hold elements of a collection, that is iterating.

Field:

→ scope and lifetime same as the object to which it is attached.

Roles:

→ helper: • hold references to objects used by many methods
• if object is passed as parameter to many methods replace it with a helper-field set in the constructor.

→ flag: • object can act in two different ways.
• if the setter-method is there, the behavior of the object can change during the life of an object.
ok: using in few conditions.
otherway use strategy field.

→ strategy: some part of an object's computation can behave in other way
• if the behavior doesn't change during the lifetime of the object, then set/pass it in the constructor.
• otherwise provide methods to change it.

→ state: like strategy-fields, but it is set within the object, not from outside.

→ components: • objects or data owned by referring object.
→ Aggregation
•

Parameters:

→ messaging between objects

→ if the same parameter is using by several methods of an object consider to attach it as a field to the referring object.

→ otherwise there is a tough coupling

→ coupling with parameters is weaker, than permanent using reference object (Helper).

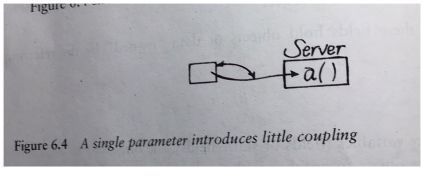


Figure 6.4 *A single parameter introduces little coupling*

leads to

Repeating the same parameter five times increases the coupling substantially:

```
Server s= new Server();
s.a(this);
s.b(this);
s.c(this);
s.d(this);
s.e(this);
```

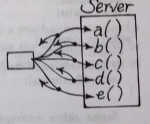
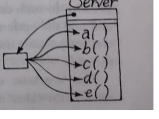


Figure 6.5 *Repeated parameter increases coupling*

In this case, the two objects are better prepared to operate independently if the parameters are replaced by a pointer:

```
Server s= new Server(this);
s.a();
s.b();
s.c();
s.d();
s.e();
```



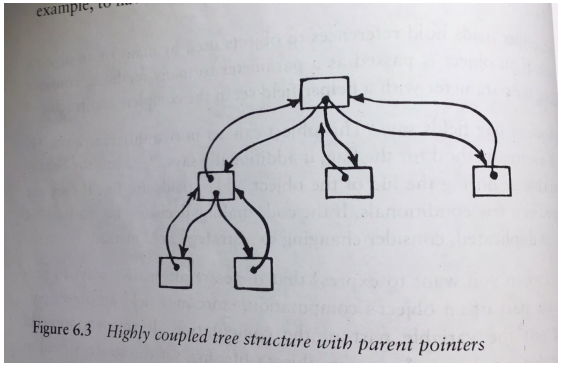


Figure 6.3 *Highly coupled tree structure with parent pointers*

- useful for algorithms like a tree, linked list.
- everywhere, where to pointers are important.

Collecting Parameter:

→ merge all return-parameters from all methods.

- if value is simple, there is no problem for example an Integer.

→ sometimes merging values is more complicated, than simple addition.

- pass the parameter, that will collect the results:

```
Node
int size() {
  int result= 1;
  for (Node each: getChildren())
    result+= each.size();
  return result;
}
```

```
Node
asList() {
  List results= new ArrayList();
  addTo(results);
  return results;
}
addTo(List elements) {
  elements.add(getValue());
  for (Node each: getChildren())
    each.addTo(elements);
}
```

Optional Parameter: default parameter!!

→ put in the interface of a method first mandatory parameters and optional parameters on the end.

- intresting:
  - → dynamic behavior / not instance-specific one
  - → flexibility
  - → default parameter

Parameter objects:

Step 1:

```
method(param1, param2, param3, param4);
```

↓ Variant A

```
method(Object object);
```

message: this list of data is strong related ←

↓ Variant B

```
method(ObjectA a, ObjectB b);
```

- data together / put the data together
- rate of change
- separation of concerns
- common state and direct access.
- fields are strong coupled.

Step 2:

→ object parameter has some state, which is using for the computation methods.

- it means you can change it! providing some new parameters.

→ expand and contract the peremeter object

- it leads to new logic, new objects, new vers of innovation.

Making the rectangle explicit as an object e

```
setOuterBounds(x, y, width, height);
setInnerBounds(x + 2, y + 2, width - 4, height - 4);
```

```
setOuterBounds(bounds);
setInnerBounds(bounds.expand(-2));
```

← object instead of params

object with extra methods

expand

Constants:

- communicate what you mean by the value
  0xFFFFFF = Color.white
- communicate variations of a message in an interface
  - setJustification(Justification.CENTER);

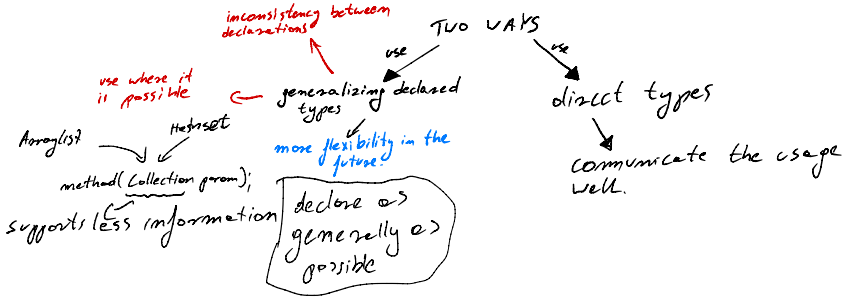
⇒ add new variants of existing methods by adding new constants without breaking implementors.

Role-Suggesting Names:

- communicate the role of a variable
- not necessary: scope, lifetime, type.

if the method is short, than the scope and lifetime is clear to read! Definition of variable not there, than it's a field.

Declared Type:



Eager Initialization:

- initialize variable as soon as it's possible

⇒ Constructor / Declaration

For symmetry ⇒ initialize everything at the same place.

Advantage ⇒ you are sure, that the variables are there, before they will be used.

No NullPointerException.

Lazy initialization:

- defer the costs of initialization.

⇒ "Performance is important"

- harder to read

⇒ it can throw NullPointerException.