Classes:

Class is a **declaration**

data & logic: should be together
change at **similar rates**

- separation of concerns
- SRP
- symmetry

- data changes much often than logic
  → the **state** of class can change regularly.

- Local consequences/changes
  - **law of demeter**
  - **tell don't ask**

class should do something significant

central concept in the design.

That's why Naming is very important.

Simple superclass Name:

- **one word name** for importing classes.
- find out the right metaphor

DrawingObject ⇒ Figure = short & punchy

→ using class names in conversations:

"Did you rotate the figure before translating it."

rotate: method
figure: class
translating: method

Balance between **length** and **expressiveness**.

length = **concise**

Qualified Subclass Name:

two jobs
"like" — "different"

→ more expressive than concise.

- subclass Name = superclass + modifiers

↳ subclass is root of hierchies.
Then use simple superclass name

Name of classes ⇒ Story of your code

Abstract Interfaces:

- old adage = code to interface, not implementation.
control of flow = DI/IoC

- design decisions = protect client from changes in implementation.
  - → not visible in more places, than necessary.
  - → I'm dealing with collections ← other code knows only this semantic invariant.

change flow of control ← visibility like design by contract

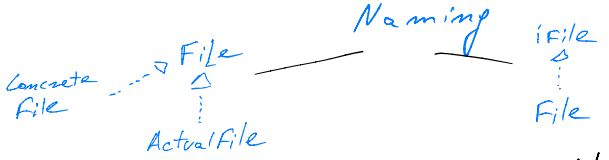
Interface: creates flexibility
Software: is unpredictable like weather.
Requirements: are not requirements
↳ semantic invariants

How to choose?

speculativ design = sth between = flexibility, when its definitely needed.
high costs — middle costs — low costs

Interface:

- protect clients from changes in the implementation.
  - ↳ local consequences
- changing interface implementation
  - ↳ global consequences = decrease flexibility
  - ⇒ all method are public for everyone.
    - ↳ why not only in a special package?



Abstract Classes:

- replace at runtime with subclass

Interface	VS.	abstract class
• here's how to access this kind of functionality.		• Here's one way to implement this functionality.
• change to an interface requires changes to all implementations. → add functionality ⇓ solution: Versioned Interface		• operations can be added without disrupting implementors

Versioned Interface:

change interface

add operation

it will break all existing implementors

- solution: declare new interface, than extend original interface and add new operations

old clients remain oblivious

old clients wants to use new operation, than downcast.

new clients can use new interface, it means:
→ old functionality/operations
→ new operations

"instanceof" = reduce flexibility by tying code to certain classes.
in this case, its justified

change interface methods
↳ renaming autocamplishon
We are not talking about it!

- interfaces are likely to change, just like all design decisions.
- learn design through implementation and maintenance.

- change of the interface structure is heavy style.

Value Object:

objects

state-changing
- → have setters
- → can change value everytime ⇒ side effects
- ⇒ for dynamic situations

value-style
- no setters = use constructor
- → never change state but create new value/object.
- → functional languages/style
  - ↳ never changing state, but creates new values.
- ⇒ for static situations
- → use "FINAL" to avoid state changing

```
class Transaction {
  int value;
  Transaction(int value, Account credit, Account debit) {
    this.value= value;
    credit.addCredit(this);
    debit.addDebit(this);
  }
  int getValue() {
    return value;
  }
}
```

Transaction will not change at some point it's state.

- no setters
- state of transaction doesn't change

```
bounds.translateBy(10, 20); // mutable Rectangle
bounds= bounds.translateBy(10, 20); // value-style Rectar
```

Specialization:

- identical logic with different input
- identical input with different logic.

- logic can be mostly same, but a little different

patterns = variety of techniques
⇒ to communicate similarity and difference of code (logic)

good expressions (symmetry) ⇒ new wave of innovation ⇒ flexibility.

Subclasses:

history:
- on the beginning subclasses were used as classifications.
  → Train subcless of vehicle.
- after a while → inheritance was using for sharing implementation

limitations:
- some set of variations isn't expressed well as subclasses than it's a tough job to restruct it.
  → code in superclasses etc.
  → difficult to understand.
- superclasses are complex ⇒ difficult to understand.
- changes to supercless are risky → subclasses rely on subtle properties from superclasses.

⇒ big and deep inheritance hierarchies

Patterns:
- use SRP in the superclasses
  ↳ overriding in the subclesses will be easier.
- every method is doing exacly one job.
- if methods are too big, then copy and change it in subclasses.

→ can't be used to express changing logic
- static binding to compile time.

Instance-specific behavior:

Theory:

1.) • all instances of a class have the same logic
→ code is completly determined by its class
= easy to read.

2.) • instances of the same class with different behaviour
→ to know, what is goimg on, you need some live examples.
= different behavior/state at the runtime.

3.) • logic changes, when computation progresses
→ algorithm behave different

Solution: • define instance-specific behavior, when the object is created.
• don't change it afterward.

Conditionals: must be local and simple.

→ if/else/switch statements are the simplest form of instance-specific behavior.

→ objects execute different logic flow based on the data.

Advantage: • logic in the same class

Disadvantage: • only change → only by code change.
• lots of paths of execution. = problem with maintenance.

→ eliminate conditional logic to messages with subclasses or delegation.

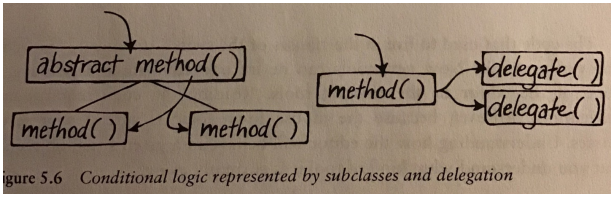


Figure 5.6 Conditional logic represented by subclasses and delegation

Delegation:

- other possibility to inheritance
- can be use for code sharing or instance specific behavior.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown(this);
}

RectangleTool
public void mouseDown(GraphicEditor editor) {
  editor.add(new RectangleFigure());
}
```

passing as a parameter
→ multiply usage
Problem ⇒ state can change at any time
global consequences/ side effects.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown();
}

RectangleTool
private GraphicEditor editor;
public RectangleTool(GraphicEditor editor) {
  this.editor= editor;
}
public void mouseDown() {
  editor.add(new RectangleFigure());
}
```

- store local as an instance attribute
→ local consequences.