

Topic 20: Further Programming

Design thinking beyond inheritance: composition, patterns, concurrency bugs and the functional style.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 20.1 Programming paradigms (object-oriented programming). Try the challenge questions on the last pages before you read the answers.

Composition over inheritance

Inheritance models an **is-a** relationship (a Teacher *is a* Person). Overused, it creates rigid hierarchies: if Duck inherits Fly() and then you need a rubber duck that doesn't fly, you have to override and disable behaviour.

Composition models **has-a**: an object is built from other objects it delegates to.

```
CLASS Duck
  PRIVATE Flying : FlyBehaviour      // a Duck HAS a way of flying
  PUBLIC PROCEDURE PerformFly()
    Flying.Fly()                    // delegate
  ENDPROCEDURE
ENDCLASS
```

Swap in FlyWithWings, CannotFly or RocketFly objects at run time without touching Duck. The design advice "favour composition over inheritance" is a common rule of thumb, and the idea of a swappable behaviour is the **Strategy** pattern.

Design patterns

A **design pattern** is a named, reusable solution to a recurring design problem. Three famous ones:

Pattern	Problem it solves	Idea
Strategy	Several interchangeable algorithms	Put each in its own class behind a common interface; the context holds one and can swap it.
Observer	Many objects need to react when one changes	The subject keeps a list of observers and notifies them all on change (button click handlers, stock tickers, spreadsheets).
Singleton	Exactly one shared instance (e.g. a settings object)	Private constructor plus a method returning the one instance. Use sparingly: it is global state in disguise.

Concurrency: race conditions and deadlock

Programs often run several **threads** at once. Trouble starts when they share data. `Count <- Count + 1` is really three steps (read, add, write), and two threads can interleave them:

```
Count = 5
Thread 1: reads 5
Thread 2: reads 5
Thread 1: writes 6
Thread 2: writes 6      <- should be 7: one update lost
```

This is a **race condition**: the result depends on timing, so bugs appear rarely and are hard to reproduce (the Therac-25 accidents in the topic 7 extension involved one). The fix is a **lock (mutex)** so only one thread at a time runs the critical section. Locks bring their own hazard, **deadlock**: Thread 1 holds lock A and waits for B while Thread 2 holds B and waits for A; both wait forever. The standard cure is for every thread to take locks in the *same order*.

A taste of the functional style

Functional programming (not in the 9618 syllabus) builds programs from **pure functions** (same inputs always give the same output, no side effects) and **immutable** data. Instead of loops that change variables you transform collections:

```
squares_of_evens = [x * x for x in data if x % 2 == 0]      # Python
total = sum(x * x for x in data if x % 2 == 0)
```

Pure functions are easy to test and reason about, and they are safe to run on many threads because there is no shared mutable state to race on: the reason the style is popular in concurrent and data-processing code.

Challenge questions

1. Say whether each relationship is better modelled by inheritance (is-a) or composition (has-a): (a) Car and Engine; (b) Student and Person; (c) a Stack class implemented using a List.
2. Two threads each execute `Count <- Count + 1` once, starting with `Count = 5`. Show an interleaving that leaves `Count = 6`, and explain how to guarantee 7.
3. Thread 1 locks A then B. Thread 2 locks B then A. Describe the problem that can occur and how to prevent it.

Worked answers

Only look once you have had a proper go at each question.

1. (a) Car **has an** Engine: composition. (b) A Student **is a** Person: inheritance. (c) A Stack **has a** List inside it (it is not a kind of list, because exposing all list operations would break the stack's rules): composition.
2. Thread 1 reads 5; Thread 2 reads 5; Thread 1 writes 6; Thread 2 writes 6. Both used the old value, so one increment is lost. Protecting the read-add-write with a lock (mutex) makes it atomic, guaranteeing 7.
3. If each thread takes its first lock before the other takes its second, Thread 1 holds A waiting for B and Thread 2 holds B waiting for A: **deadlock**. Prevent it by making all threads acquire locks in the same global order (always A then B).