SoC *(Separation of Concerns)*: Är ett bredare koncept som innebär att separera programmet i olika "bekymmer" eller delar (t.ex. gränssnitt, affärslogik, databas). Det gör det möjligt att arbeta med enskilda delar isolerat.

SRP *(Single Responsibility Principle)*: Är mer specifikt och appliceras oftast på klasser, funktioner och moduler. En klass ska bara göra en sak och vara ansvarig mot en enda aktör.

Uppgift 3

AbstractCar – SRP: OK, men risk för överbelastning

Ansvar:

- Hålla bilens state (speed, direction, x,y)
- Implementera grundläggande rörelse
- Validera gas/brake

Problem:

Klassen ansvarar både för movement och speed clamping.
Den har direkt åtkomst till position (x,y) och direction, vilket är OK, men gör det svårare att isolera movement.

Inte akut med förändring.
Men man kan överväga att bryta ut movement-logiken till en separat strategi längre fram.

Volvo240 / Saab95 / Scania – SRP OK

Dessa är rena specialiseringar, följer SRP.

Problem i Scania:

Metoden flank(int v) blandar validering, logik för att ändra vinkeln, och felhantering.
Kan delas upp i:

- raiseBed(...)
- lowerBed(...)
- intern validering.

CarTransport – bryter mot SRP

Ansvar:

- Är ett fordon
- Hanterar ramp
- Hanterar last
- Har FILO-stack
- Har positionsuppdatering av lastade bilar
- Har avståndsvalidering
- Har förhindrande av körning när ramp nere

Problem:
4 olika ansvar i samma klass, bryter SRP rejält.

Borde delas i:

- CarTransport (fordonet)
- LoadingPlatform (ramp + constraints)
- CargoBay (lastning/losning, FILO-stack)
- DistanceChecker (validering)

Workshop – SRP OK, men enkel
Litet och tydligt ansvar: lagra bilar.

Problem:
Tar inte själv hänsyn till positioner (det gör CarController).
Beroendet bör vändas så att Workshop kan ta beslut om bilar kan tas emot.

CarController – Sämst SRP
Den här klassen har stora ansvarsproblem.

Den gör följande:

- Uppdaterar GUI
- Uppdaterar modellens state
- Kollisionsdetektion mot väggar
- Kollisionsdetektion mot workshop
- Uppdaterar positionsdata i DrawPanel
- Kör timerloop
- Tar hand om knapptryckningar från view

- Hanterar turbo
- Hanterar Scania flak
- Hanterar lastning i verkstad
- Start/Stop motorer
- Lagrar en bil→id mapping
- Lagrar även workshop och dess position

Bör delas upp i minst:
- CarManager (modellhantering)
- CollisionSystem (väggkollisioner)
- SimulationLoop
- Controller
- CarPositionMapper

CarView

Ansvar:
- Renderer
- UI-layout och alla knappar
- Eventkoppling

Det är OK, men kopplingen till CarController är hård.
Man skulle kunna introducera att interface istället.

DrawPanel

Ansvar:
- Rendera bilder
- Hantera bilarnas positioner
- Lagra images
- Flyttlogik

Borde ändras till:
- Bara hantera rendering
- kanske borde ta emot data med bilarnas positionsvärden

Uppgift 4

Refaktoreringsplan

Steg 1 – Introducera ICarController
- Skapa interface
- Låt CarView ta emot interface istället för CarController
- Kan göras parallellt.

Steg 2 – Skapa CarManager
- CarController flyttar ut alla bil-listor till CarManager
- CarManager ansvarar endast för:
- addCar/removeCar
- getCars()
- updateAllCars()
- Kan göras parallellt.

Steg 3 – Skapa MovementSystem
- Flytta rörelselogik till separat klass
- CarController anropar movementSystem.update()
- Kräver samordning med steg 2.

Steg 4 – Skapa CollisionSystem
- Flytta väggkollisioner till separat klass
- CarController kallar collisionSystem.check()
- Kan göras parallellt efter CarManager är klar.

Steg 5 – Flytta workshop-kollisioner
- Flytta WorkshopCollision från CarController in i WorkshopCollisionSystem
- Kan göras parallellt.

Steg 6 – Skapa CarDTO och uppdatera DrawPanel
- DrawPanel tar endast emot CarDTO[]
- Tar bort all positionsdata
- Kräver samordning mellan view och controller.

Steg 7 – Refaktorera CarTransport
- Dela upp i:
- Ramp
- CargoBay
- CarTransport
- Görs sist