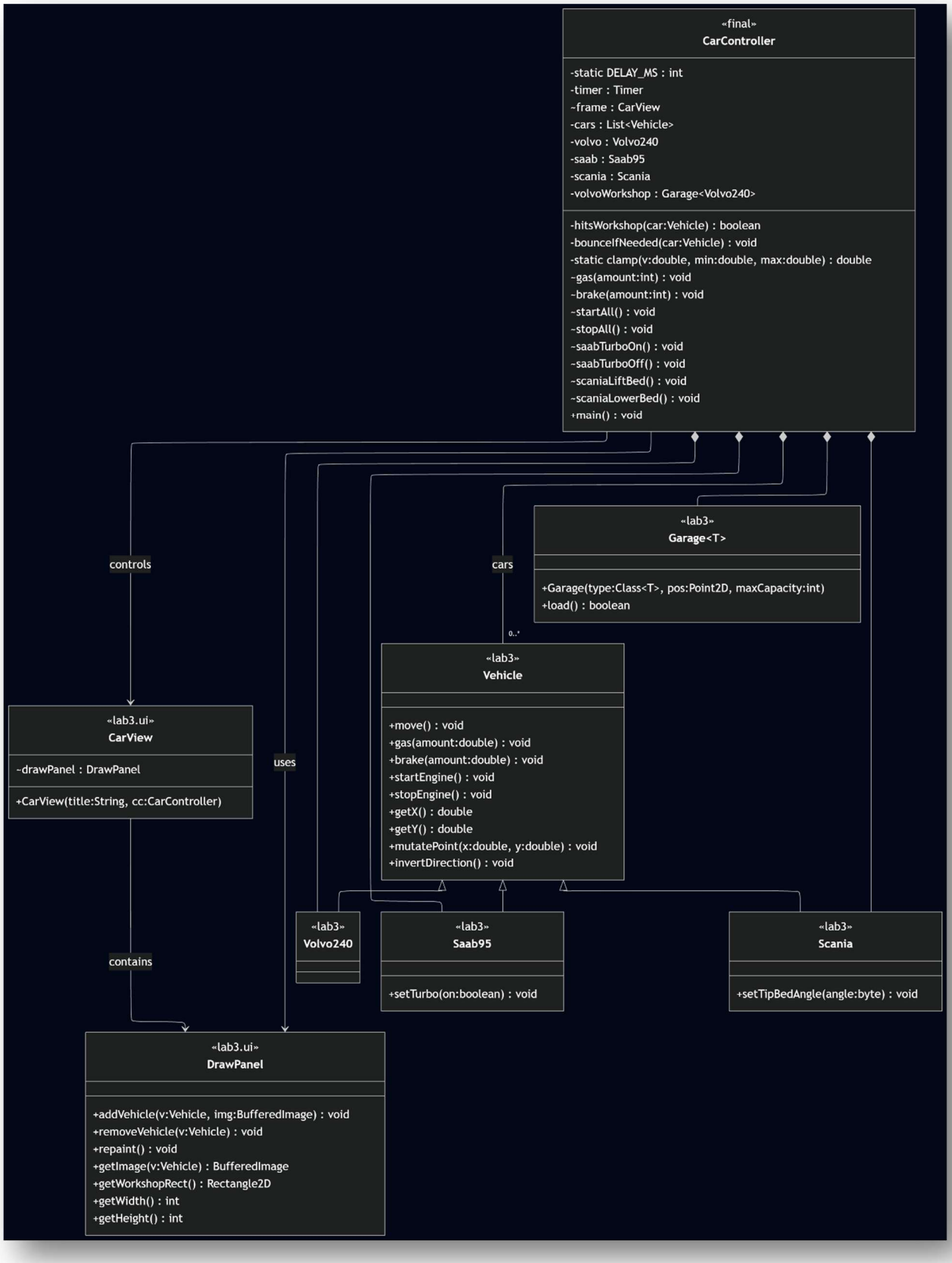
Utfärdare: Durim Miziraj
Kontakt: gusmizdu@student.gu.se

Laboration 3: Design och principer



Utfärdare: Durim Miziraj
Kontakt: gusmizdu@student.gu.se

Uppgift 2: Beroenden

Det finns problem med sammanhållningen i en majoritet av klasserna. Nästan alla klasser hanterar beteenden och tillstånd som konceptuellt inte hör till dem. Det jag tänker på främst är koordinatsystem och funktioner för krockigenkänning som är spridda lite överallt. Vid första antagande kan man lätt tänka sig att position och visuella avgränsningar hör till objekten som position och avgränsning berör. Den förvillelsen gör att det ser ut på det här viset. Det jag borde haft är ett starkare koncept av ett ”världsobjekt”, där allt som berör ”världen” samlas.

Det världobjektet hade fått ägas av en annan klass som hade förbättrat objektens konceptuella integritet (sammanhållning). Den klassen hade fått heta `'DomainModel'`. Den hade rensat upp på både 'DrawPanel' och `'CarController'` genom att ta bort ansvaret för Timer och andra motor relaterade funktioner. Den hade också kunnat leverera oföränderliga ”snapshots” till `'CarView'` som i princip inte borde göra något annat än att måla efter instruktioner som den fått av ”världsobjektet” via `'DomainModel'`.

I och med införandet av `'DomainModel'` så minskar även graden av ”coupling” i koden. Här hade `'CarController'` och `'CarView'` fått varsin referens (via ett interface) till `'DomainModel',` som i sin tur har readonly API'er för `'CarView'` som läser ”snapshots” och `'CarController'` som skickar kommandon.

För att få till detta så behöver `'CarController'` fråntas ansvaret som programmets startpunkt. Detta gör jag enkelt genom en Main klass som instansierar model, view, controller och förmedlar de referenser som behövs.

Konkreta svar.

- Vilka beroenden är nödvändiga?

Det enda nödvändiga beroendet är mellan den abstrakta `'Vehicle'` klassen och dess arvtagare. `'CarController'` och `'CarView'` beroendet är också nödvändigt ifall man förutsätter en MVC arkitektur.

- Vilka klasser är beroende av varandra som inte borde vara det?

Alla beroenden hos `'CarController'`, förutom `'CarView'` är onödiga.

- Finns det starkare beroenden än nödvändigt?

Beroendet mellan `'CarController'` och de konkreta fordonsklasserna. Och ifall man enbart förlitar sig på `'Vehicle'` för att komma åt beteendena hos fordonen så skulle det beroendet också vara för starkt.

- Kan ni identifiera några brott mot övriga designprinciper vi pratat om i kursen?

Tror inte att vi har tagit upp detta, men `'Car'` interfacet definierar inga beteenden, den står helt tom.

Utfärdare: Durim Miziraj
Kontakt: gusmizdu@student.gu.se

Uppgift 3: Ansvarsområden

Som nämnts ovan så är ansvaren för breda i en majoritet av klasserna. Den mest uppenbara bristen på SRP är klassen `'CarController's` roll som programbyggare.

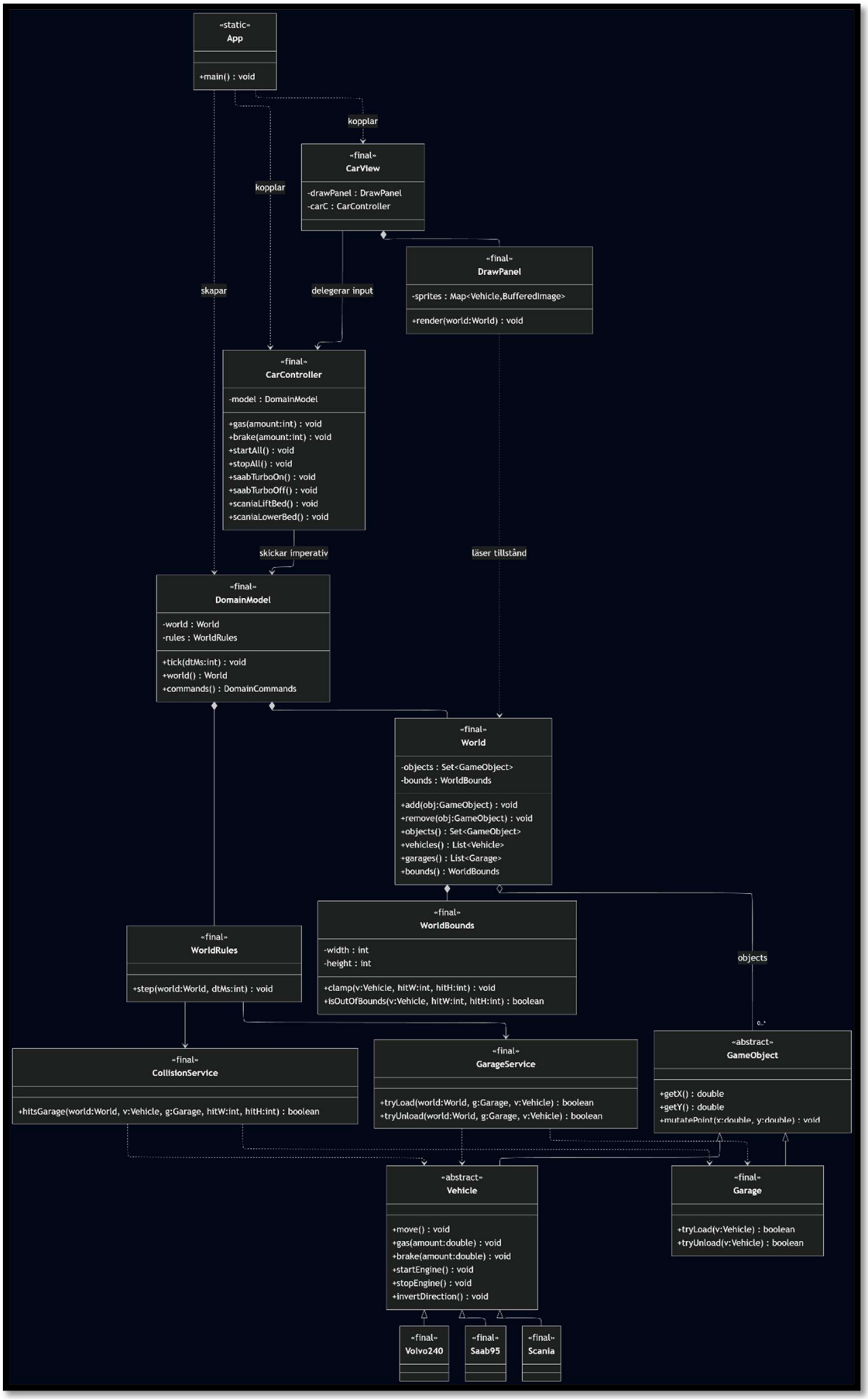
Bristen på ett centralt koncept av ”värld” har skapat dubbla sanningskällor i `'GameObject'` och `'CarController'`. Ett parktexempel på vad som kan gå fel när SoC inte följs.

- Vilka ansvarsområden har era klasser?
    - CarContoller – Tar domänansvar såväl som ansvar för delegering av användarkommandon.
    - CarView och DrawPanel – Bygger UI och målar upp scenen.
    - Vehicle + subklasser – Beteenden och tillstånd (state) för alla fordonen.
- Vilka anledningar har de att förändras?
    - CarController – Tillökning av funktioner i domänmodell kräver förändring i klasser som kontrollerar den.
    - CarView och DrawPanel – UI komponenter läggs till/tas bort.
    - Vehicle + subklasser – Beteenden och tillstånd (state) läggs till eller tas bort.
- På vilka klasser skulle ni behöva tillämpa dekomposition för att bättre följa SoC och SRP?

Mest kritiskt är `'CarController's` översträckta ansvarsområde och listan med objekt i `'DrawPanel'`. Dessa klasser ska inte göra något annat än måla upp domänet och skicka imperativ till det, efter andras instruktioner.

Utfärdare: Durim Miziraj
Kontakt: gusmizdu@student.gu.se

Uppgift 4: Ny design



Utfärdare: Durim Miziraj
Kontakt: gusmizdu@student.gu.se

Motivation

I mitt ändringsförslag tar `'CarController'` inget annat ansvar än att ta imperativ från UI-lagret och delegera dem till `'DomainModel'`. Detta följer SoC)genom att UI-interaktion separeras från domänlogik. Det förbättrar också SRP eftersom `'CarController'` nu endast har en tydlig anledning att ändras. Dvs om UI-kommandon eller deras mappning ändras.

DomainModel fungerar som en orkestrerare för domänlagret. Den inkapsulerar hur världen uppdateras och exponerar vad som är säkert en säker sanningskälla. Detta minskar kopplingen mellan UI och domänens interna struktur och är i linje med Law of Demeter. Dvs att UI/Controller inte behöver navigera runt i många objekt för att utföra en operation.

World är den centrala sanningskällan för världens tillstånd (state) och håller endast `'Set<GameObject>'` objekt. Detta tar bort överflödiga representationer som separata listor för fordon och garage. Det minskar risken för inkonsistens och förbättrar både robusthet och cohesion.

WorldRules beskriver hur världen får förändras. Genom att flytta regler som kollision, avgränsningar och lastning/lossning från UI-lagret till `'WorldRules'` placeras regler där domänkunskapen hör hemma. Detta bidrar också till OCP.

Refaktoriseringsplan

Länken Övning 3: UML, static vs dynamic skickar mig till en stängd sida. Jag har inte kunnat ta del av exemplet.

Steg 1. Flytta programmbygget från `'CarController'` till en `'Main'` klass.

Steg 3. Introducera `'DomainModel'` och flytta dit timer och andra domänrelaterade ansvar.

Steg 3. Introducera `'World'` och `'WorldRules'` och sätt dit ”världslogik”.

Steg 4. Skapa interface kontrakt och kanske en JSON objektmodell som ska agera som prototypdata mellan världstillstånd och UI.

Steg 5. Skapa ett öppet API i `'DomainModel'` och avsläsning samt uppmålning av ”världen” i UI lagret. Två utvecklare kan paralellt arbeta här.

Utvecklare 1 sätts på steg 1 till 3.

Steg 4 utförs i samråd mellan utvecklare 1 och 2.

Steg 5 kan utföras paralellt mellan utvecklare 1 och 2.