Uppgift 3:

Observer, Factory Method, State, Composite. För vart och ett av dessa fyra designmönster, svara på följande frågor:
Finns det något ställe i er design där ni redan använder detta pattern, avsiktligt eller oavsiktligt? Vilka designproblem löste ni genom att använda det?

Finns det något ställe där ni kan förbättra er design genom att använda detta design pattern? Vilka designproblem skulle ni lösa genom att använda det? Om inte, varför skulle er design inte förbättras av att använda det?

Uppdatera er design med de förbättringar ni identifierat.

Observer Pattern:
Detta designmönster använde vi i integrationen mellan Widget och CarController. Från början var CarView och CarController beroende av varandra, båda klasser behövde en instans av den andra klassen. Dessutom innehöll CarView en del logik som egentligen borde höra till kontrollern, vilket bröt mot OCP eftersom en förändring av ena klassen skulle behöva leda till en förändring av den andra klassen. Ett ytterligare problem var att view inte var uppdelat från controller enligt MVC. Dessutom innehöll CarView widgets vilket strider mot MVC. För att lösa detta skapade vi först en ny klass vi kallar för Widget, där alla knappar och dylikt istället ligger. Vi införde ett interface mellan Widget och CarController som heter WidgetListener som sedan implementeras av CarController. Widget tar in en instans av en WidgetListener vilket innebär att den kan broadcasta sina event till vem som helst som vill lyssna. Dessa förändringar bryter beroendet mellan CarView och CarController och därav löser problemet med OCP. Flytten av en del funktionalitet från CarView till Widget löste problemet med MVC. Vi anser inte att ytterligare användning av Observer skulle förbättra vår design.

Factory Pattern:
Som en del av vår refaktoriseringsplan hade vi identifierat att vår view och controller (efter vår refaktorisering nu applikation) var väldigt beroende på de konkreta implementationerna av klasserna i modellen, dvs klasserna som utgör fordonen vilket bryter mot DIP. För att lösa detta skapade vi en ny klass som heter Factory som applikationen istället kan kalla på för att skapa bilar. Vi anser inte att en extra Factory skulle hjälpa designen, snarare stjälpa.

State Pattern:
Det är inte ett designmönster som använts i vår kod. Det tydliga, potentiella användningsområdet skulle vara i samband med Flatbed, som då skulle kunna ändras mellan två states, raised och lower. I vår nuvarande implementation, använder Flatbed och Platform samma interface Attachment, vilket då inte skulle gå att göra längre för att Platform har flera olika angles. Då skulle möjligheten vara att göra ett AttachmentState-interface som båda använder där sedan ett ytterligare interface AngledAttachmentState eller dylikt där Platform-angle-logik får implementeras. Detta kräver dock typecasting i vår Truck-class i dess nuvarande skick, vilket gärna undviks. Vi ser inte att det förbättrar koden då det är en enkel implementation av Flatbed i nuläget som inte behöver förbättras.

Composite Pattern:
Detta mönster används oavsiktligen mycket i vår kod då vi ofta loopar igenom listor av bilar och utför samma operation för dessa. Det löser ett problem där flera objekt behöver följa samma beteende trots

att de kan vara av olika typer. Tydligast är detta i CarTransporter där alla loadedCars förflyttas tillsammans med transportören och dess loadable. Alla bilar som lastats behandlas enhetligt och de hanteras som en grupp. I övrigt ser vi inga nödvändiga användningsområden för Composite Pattern.