2B

2.

- Vyn borde ha gjorts dummare, controllern borde vara tunnare och modellen smartare. De hade inga väl avgränsade ansvarsområden.
- Vi valde att följa MVC-pattern, alltså gjorde vi vyn dum osv samt gjorde så att de fick väl avgränsade ansvarsområden. En möjlig brist är att DrawPanel fortfarande är beroende på CarModel.

3.

- Då vi redan använder oss av Observer-pattern så kan en utökning med denna nya vy ske utan modifikation genom att implementera vårt observer-interface.
- Genom Observer-pattern:s “notify” metod sker kommunikation mellan komponenter.
- Har så gott det går gjort att alla klasser i MVC förutom CarModel har beroende på Vehicle eller någon av dess subklasser

4.

- **Observer:**
  - Används i vår MVC för att få våra views att uppdateras när modellen ser detta lämpligt. Genom detta är det möjligt för oss att implementera fler olika vyer och det åstadkoms lösare beroenden.
  - Skulle möjligen kunna använda detta pattern i vår DrawPanel också. Dock används detta pattern i CarView för att få just DrawPanel att uppdateras då DrawPanel är en instans i CarView. Detta är dock starkare beroenden än vad som kanske skulle behövas, till exempel om DrawPanel istället implementerar vårt observer-interface. Dock finns DrawPanel just nu i Carview för att få den att hamna på rätt ställe i vyn vilket den annars inte gör på grund av ordning som alla komponenter instansieras i.
  - På grund av det sistnämnda bedöms det inte vara fördelaktigt för vår design att implementera Observer-pattern även för DrawPanel.
- **Factory method**
  - Detta pattern använder vi oss inte av i nuläget.
  - Skulle kunna använda detta pattern vid skapande av bilar (eller vid skapandet av vyer?). På så sätt dölja mer av den interna implementationen. Bäst hade varit att i såfall följa Factory-pattern. Skapar förutsättning för att kunna dölja bilklasser i ett package.
- **State**
  - Används inte just nu.
  - Kan användas i Saab-klassen för att hålla koll på om turbon är av eller inte. Hade inte spelat någon större roll för implementationen då inga andra klasser skulle beröras av förändringen. Bör dock genomföras för elegans.
- **Composite**
  - Används inte just nu.
  - Kan användas för bilar. Måste dock ha olika Composite klasser för Car och Vehicle då vi annars stöter på problem med bland annat Transporter/Transportable funktionaliteten. Denna samlingsklass för Car kan då även implementeras i Transporter-klassen samt i Workshop istället för att använda listor. Composite-klassen för Vehicle kan användas i vår CarModel

klass. Med detta pattern skulle vi på så sätt följa Separation of Concern-principle bättre.
    - Då varje instans av de olika bil-klasserna har en intern representation av dess position som är olika för olika bilar kommer en instans av Composite klassen inte kunna ha till exempel getters för detta. Detsamma gäller för getter för saker såsom färg, currentSpeed m.m. Att därmed implementera Composite pattern betyder att vi inte kommer kunna behandla det precis som en enskild instans. Kommer då behöva göra många avvägningar för olika metoder vilket leder till mindre elegans i vår mening.
    - Överlag verkar en refaktorering för att använda Composite-pattern medföra fler problem än vad det tillför gott till implementationen.

7.

- Vi anser att bilarna i vår implementation borde vara muterbara och har därav valt att inte göra dessa klasser “immutable”. Getters i Vehicle-klassen returnerar även värden av primitiv typ.