TDDE10
725G90

Objektorienterad programmering i Java
Föreläsning 9

Emma Enocksson Svensson & Magnus Nielsen

LINKÖPINGS UNIVERSITET

Anmäl er till tentan!!!!!

Anmälan öppnar på måndag 19/2

Dagens föreläsning

- Objektorienterad analys
- Objektorienterad design
- Strömmar
- Filer
- Undantag
- Generiska klasser
- Nästlade klasser
- Debugger

Ett något större exempel

- En tekniker på en verkstad behöver ett system för att hålla reda på fordon, dess ägare och vad som behöver servas (beroende på typ/modell)

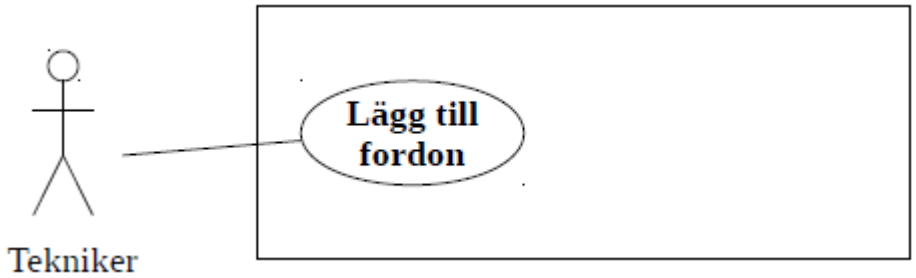
- Systemkrav (ett exempel):
  - Systemet ska hålla reda på fordon och dess ägare
  - Teknikern ska kunna lägga till och ta bort fordon
  - Systemet ska, för ett fordon, kunna generera en checklista över det som skall servas

Nu vet vi hur man bygger klasshierarkier, ritar UML och programmerar objektorienterat.

Men hur kommer vi fram till vilka klasser som behövs och vilket ansvar de skall ha?

Användningsfall (eng. Use Case)

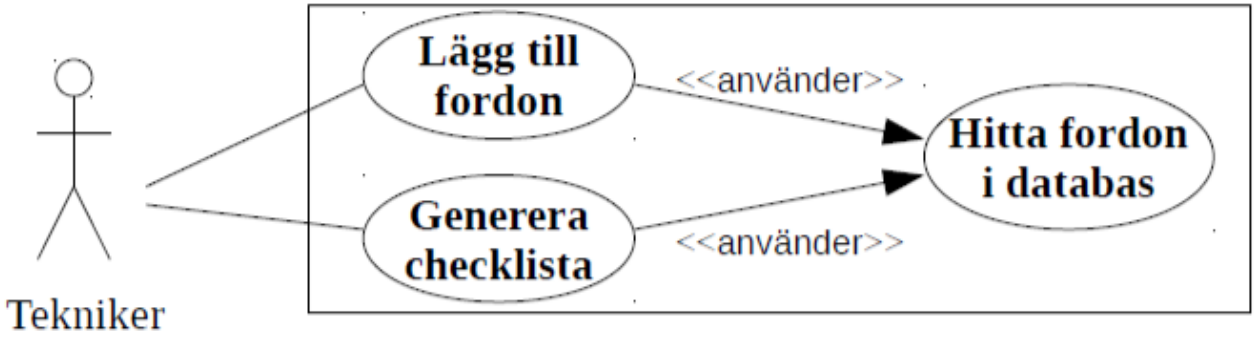
- Beskriver en interaktion mellan en användare (en aktör) och systemet. Aktören är ofta en människa, men behöver inte vara det. Det kan t.ex. vara ett annat system
  - Utgör en funktion som är utåt synlig
  - Uppnår ett distinkt mål för användaren
  - Omfattning kan variera
  - Innefattar en bild och beskrivande text (Use-case diagram)



Teknikern lägger till ett fordon till systemet genom att fylla i ett formulär. När teknikern klickar på knappen ”bekräfta” kontrolleras att fordonet inte redan finns i systemet och att det finns i transportstyrelsens register. Skulle något av dessa gälla visas ett felmeddelande, annars läggs fordonet till.

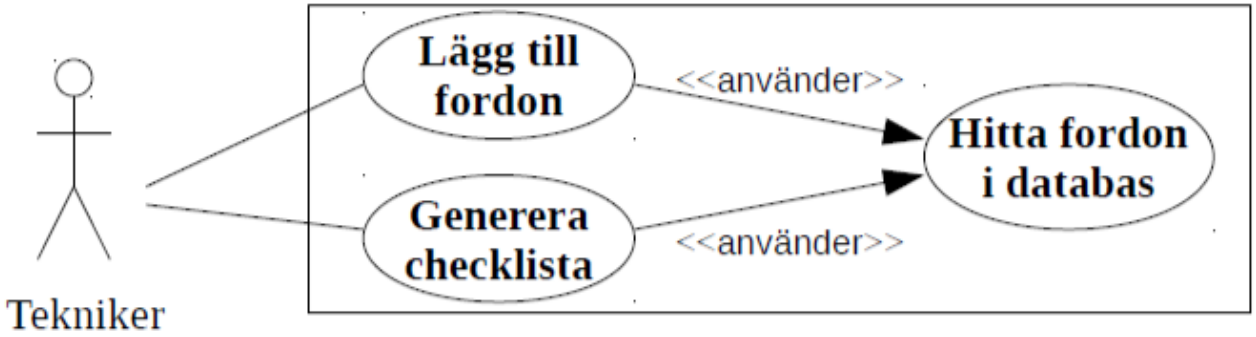
Användningsfall (eng. Use Case)

- Gemensam funktionalitet kan brytas ut
- Relationen *använder* används till att referera till den gemensamma delen.



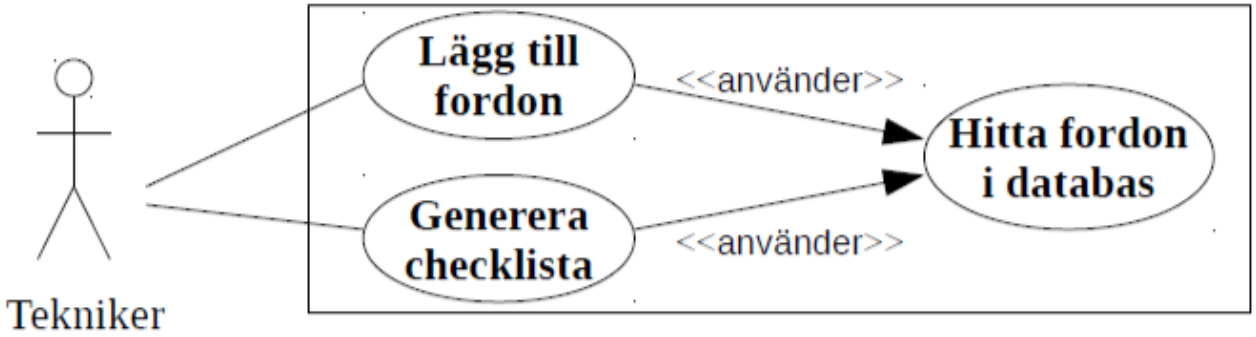
- Finns även relationen utvidgar som används för att specialisera ett mer generellt användingsfall
  - T.ex ”Ta bort fordon ur databas” skulle kunna utvidga ”hitta fordon i databas”

Användningsfall (eng. Use Case)



- En viss väg genom ett användingsfall kallas för ett *scenario*
  - T.ex. *Teknikern matar in fordonets registreringsnummer och klickar på ”hitta fordon”. Informationen för en MC dyker upp på skärmen. Därefter klickar teknikern på ”generera checklista”. På skärmen dyker det upp en lång lista med olika rubriker (checklistan)*

Användningsfall (eng. Use Case)



- Kan vi redan nu identifiera några klasser/objekt?
  - *”Leta substantiv”*
  - *Brainstorming*

Fordon

MC

Registrerings-
nummer

Lastbil

Fönster

Checklista

Knapp

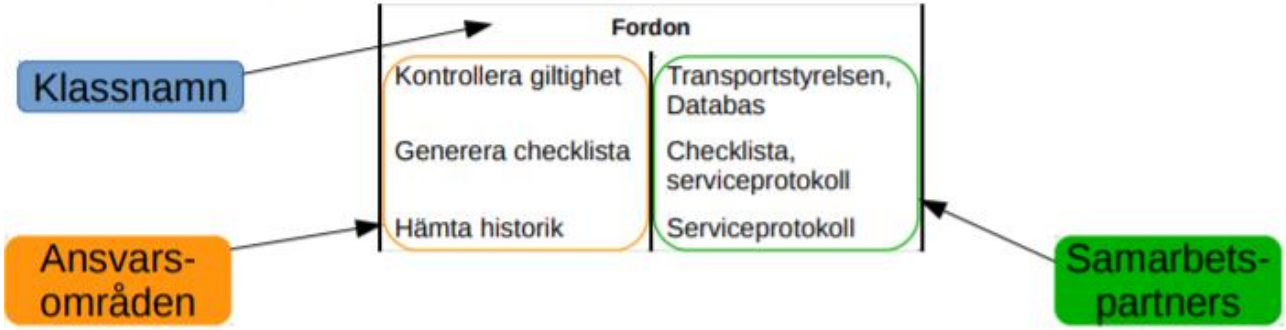
Transportsstyrelsens register?

Personbil

”Databas”

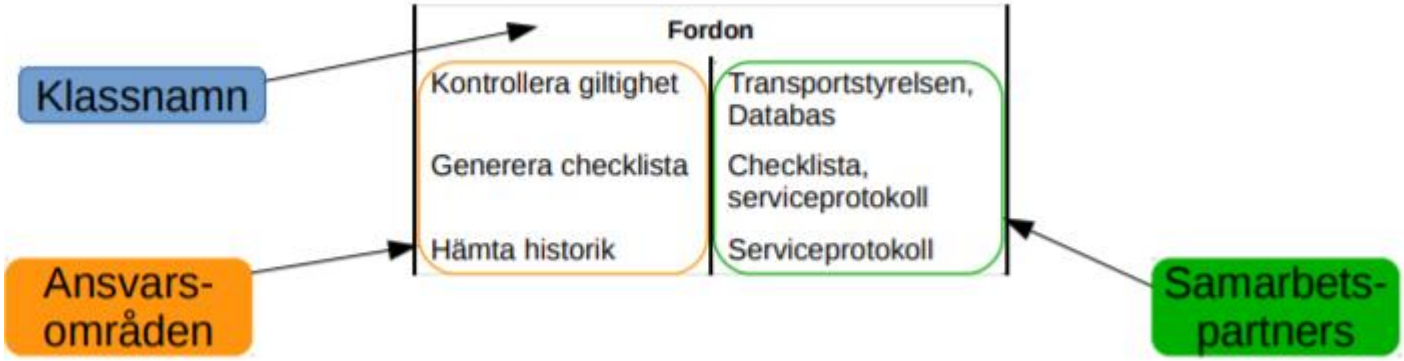
Klass-kort (CRC-kort)

- VI kan använda oss av Klass-kort för att se våra klasser och dess samband
- CRC står för Class, Responsibilities and Collaborators. Det innebär:
  - Vad har vi för klasser?
  - Vad är deras ansvar?
  - Vilka klasser samarbetar de med?
- Kan användas tillsammans med, eller som grund till klassdiagram



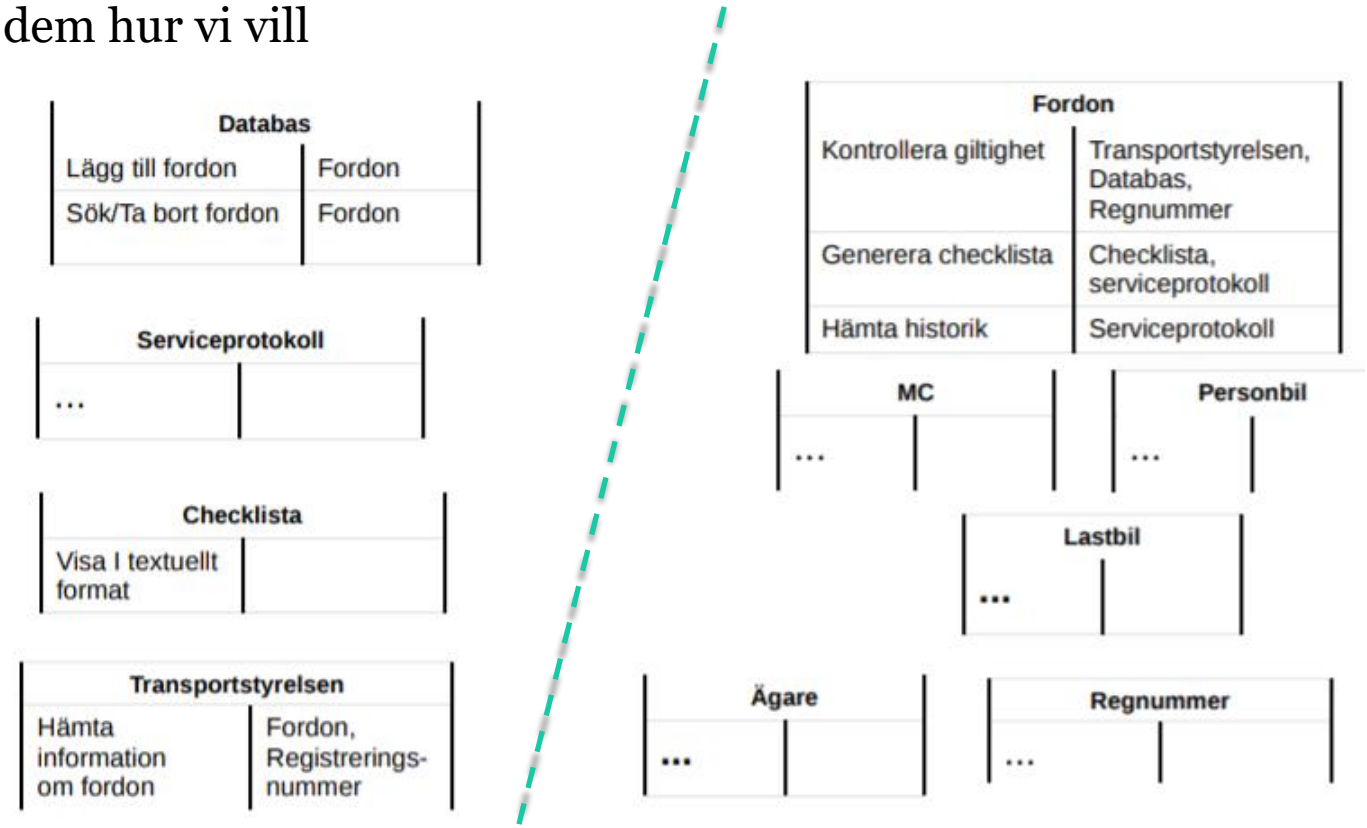
Klass-kort (CRC-kort)

- Just ”ansvar” för en klass är viktigt att fundera på
  - *Leder bort från tänket att klasserna är passiva databehållare*
  - *Underlättar för förståelsen om klassens beteende i stort*



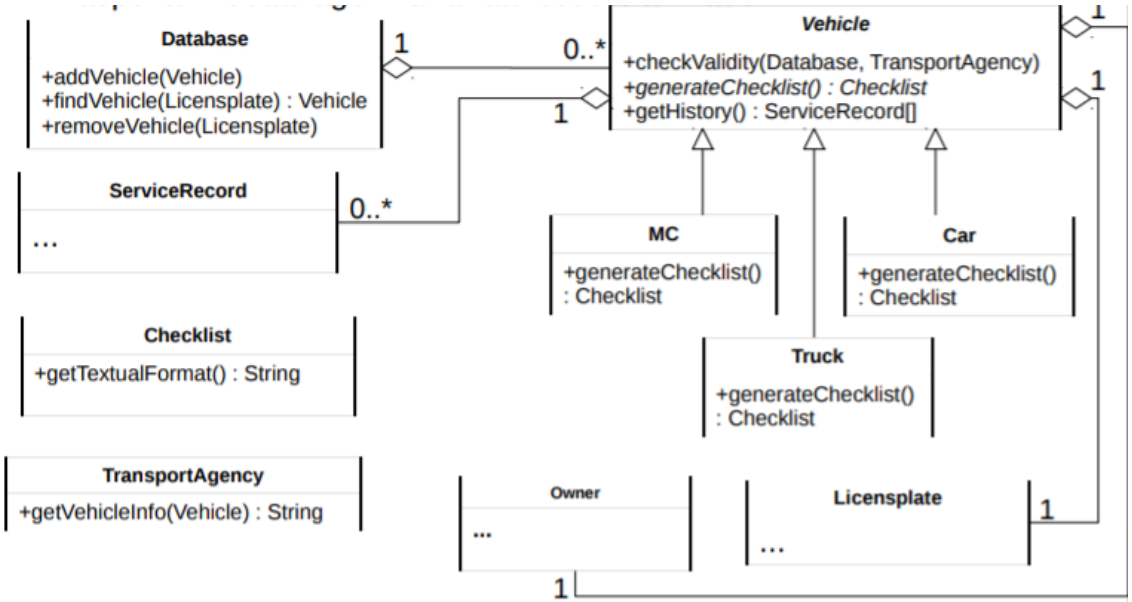
Klass-kort (CRC-kort)

- Vi kan nu skapa CRC-kort för våra klasser och dra runt och gruppera dem hur vi vill

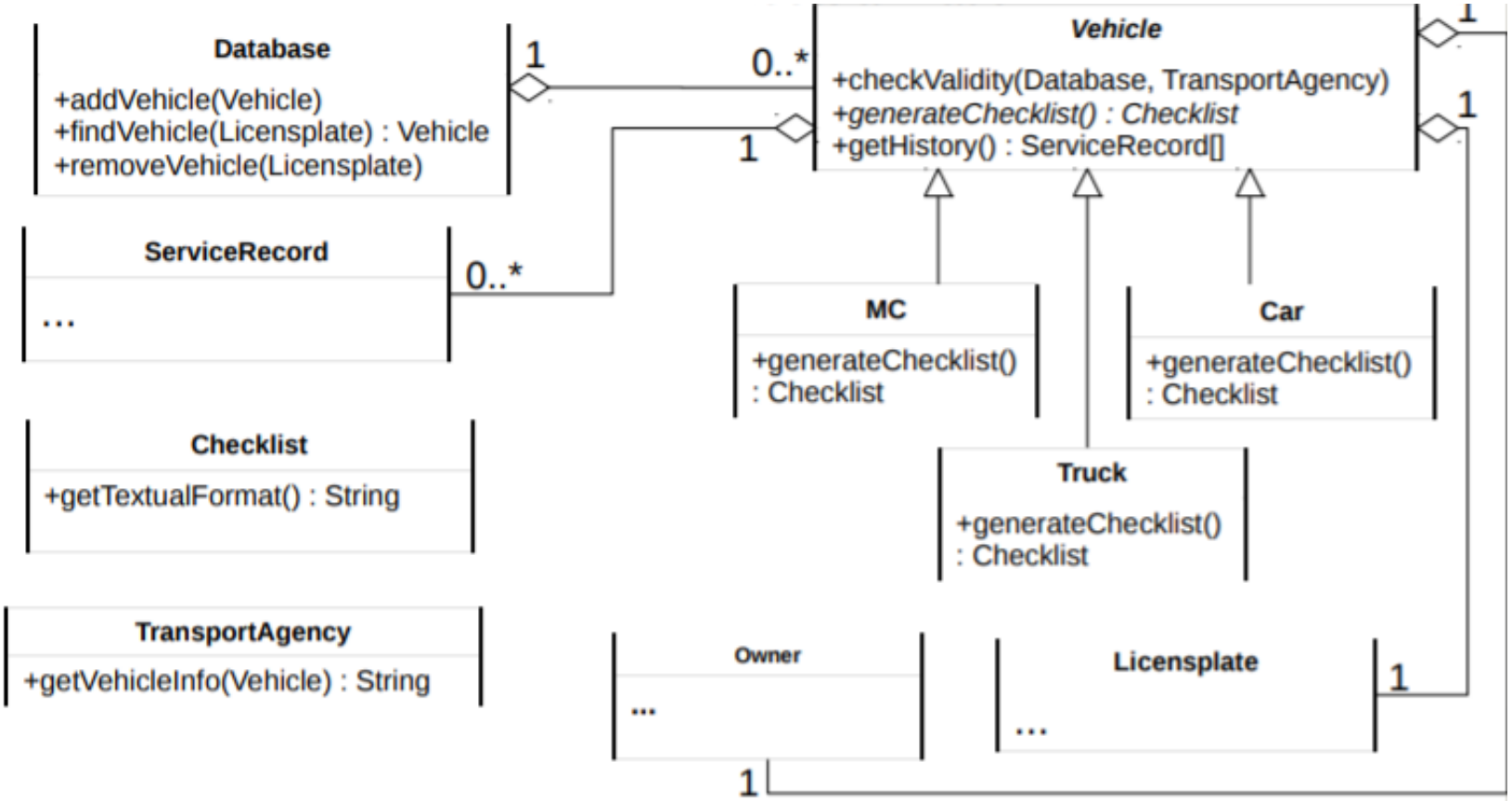


Klassdiagram

- Slutligen bör vi även beskrivna relationerna mellan våra funna klasser. Här använder vi gärna klassdiagram
  - Diagrammet behöver inte vara jättedetaljerat ur andra aspekter i detta läge. Här är attribut utelämnade

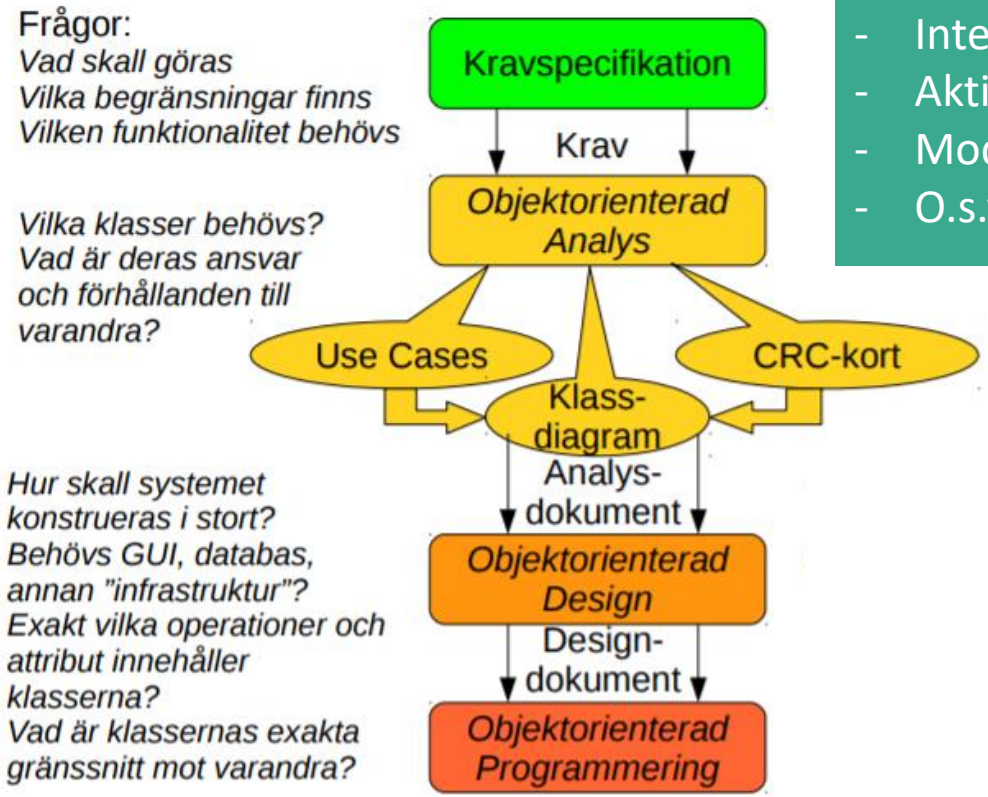


Klassdiagram



Ett något större exempel

- Vi höjer blicken lite. Vi fick lite givna krav



Finns självklart fler verktyg!
- Interaktionsdiagram
- Aktivitetsdiagram
- Moduldiagram
- O.s.v.

Objektorienterad Design

- I designdokumentet förfinas klassdiagrammet och klassbeskrivningarna. Även Use cases, scenarion och andra diagram som tagits fram i analysfasen finslipas.
- Ett utdrag från ett färdigt designdokument:

Klass *Vehicle*

Beskrivning

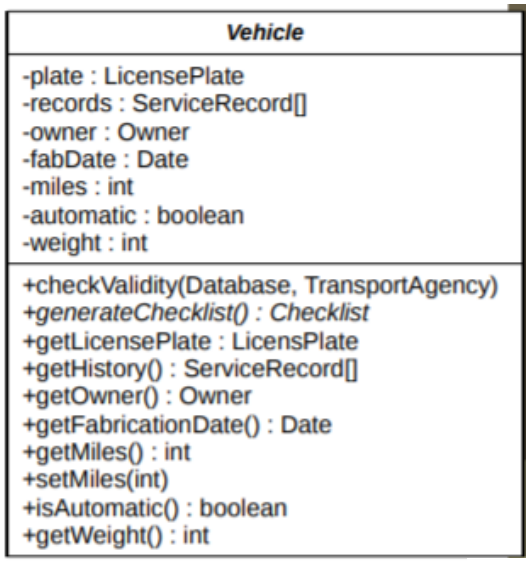
Klassen Vehicle är superklassen för alla fordon I systemets databas. Klassen lagrar data för ett fordon och ansvarar för giltighetskontroller av fordonet samt generering av checklista. Klassen är abstrakt, eftersom metoden generateChecklist skall överskuggas i basklasserna Car, MC, o.s.v.

Konstruktorer

Det finns endast en konstruktor som ansvarar för att sätta alla klassens instansvariabler från inkommande parametrar (en per instansvariabel).

Publika instansmetoder

...



Den sista javan för kursen

Strömmar, Filer, Undantag, Generiska klasser, Nästlade klasser, Eclipse Debugger

Strömmar

Definition: En ström (eng. Stream) är en sekvens av data från någons källa och/eller till något mål.

- In och utmatning i Java utförs av strömmar
- Vanliga exempel inkluderar:
  - System.out

    Vanligen kopplad till terminalen
    Av klassen *PrintStream*
  - System.err

    Vanligen kopplad till terminalen
    Av klassen *PrintStream*
  - System.in

    Vanligen ihopkopplad med annan ström som **formaterar indatat**, ex. *BufferedREader, BufferedLineReader, Scanner*
    Av klassen *InputStream*

Strömmar

Exempel

```
// Kopplar ihop ett reader-objekt med in-ström
BufferedReader input = new BufferedReader(
                        new  InputStreamReader(System.in));

// läser en rad och konverterar till int
int num = Integer.parseInt(input.readLine());
…
```

Scanner

En praktisk klass för att ”parsa” datat som kommer på en ström.

```
// Kopplar ihop ett reader-objekt med in-ström
Scanner keyboard = new Scanner(System.in);
// läser en rad och konverterar till int
int num = keyboard.nextInt();
double d = keyboard.nextDouble();
String word = keyboard.next();  // separerat med vita tecken
String line = keyboard.nextLine();  // separerat med vita tecken

// egentligen borde man kolla att det finns data att läsa…
if (keyboard.hasNext()) {
  // ok att läsa
}
```

Scanner

Kan hantera alla inströmmar, och även strängar

```
String data = ”6 laxar i 1 lax-ask”;
Scanner scan = new Scanner(data);

scan.nextInt(); // läser 6
scan.next();    // läser ”laxar”
scan.nextInt(); // KRASCH! ”i” kan inte tolka som int!
```

Formatering

Ofta vill man skriva ut på en viss form

```
double d = 10.0/3.0;
System.out.println(d); // 3.3333333333333335
```

Metoden format i PrintStream kan hjälpa till:

```
System.out.format("Värdet %1.2f är bra grejor.\n", d);
                    // Värdet 3,33 är bra grejor.
Locale.setDefault(Locale.US);
// Ändra format enl. ”lokalen”
System.out.format("The value %1.2f is indeed good.\n", d);
                    // The value 3.33 is indeed good.
```

Formatering

Formatspecificerare är på följande form:

%[flagga][bredd]typ

Typ	Förklaring	Flaggor	Förklaring
d	Heltal, decimal form	-	Vänsterjustering
x	Heltal, hexadecimal form	+	Talets tecken skrivs ut
f	Reellt tal, decimalform	*blankt*	Positivt tal inleder med blanksteg
e	Reellt tal, exponentform	,	Siffrorna grupperas tre och tre

Fler specificerare finns, se Javas API-dokumentation

Filer

Hantera själva filen med klassen File.
Filer läses som strömmar.
För textfiler, använd FileReader resp. FileWriter.

```
String filename = "myfile.txt";
File file = new File(filename);
FileReader reader = new FileReader(file);
Scanner scanner = new Scanner(reader);
//...
scanner.close();     // bra att stänga strömmar när
     // man är klar med dem.
scanner.nextInt();   // error, stängd ström.
```

Filer

Magnus	19
Emma	20
Simon	16
Martin	15

Exempel (Läsa in löner till en map)

```
Scanner scanner = new Scanner(new File("wages.txt"));
Map<String, Integer> map = new HashMap<>();

while (scanner.hasNext()) {
  String name = scanner.next();
  int wage = scanner.nextInt();
  map.put(name, wage);
}
System.out.println("Magnus har " + map.get("Magnus") +
                   " kr i lön.");

scanner.close();
```

Kan kasta FileNorFoundException

Filer

Exempel (Skriva till en textfil)

```
try {
  FileWriter writer = new FileWriter(new File("text.txt"));
  writer.write("Stuff\n");
  writer.close();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

Filer

Exempel (Skriva till slutet av en textfil)

```
try {
  FileWriter writer = new FileWriter(new File("text.txt"), true);
  writer.write("Stuff\n");
  writer.close();
} catch (IOException ioe) {
  ioe.printStackTrace();
}
```

på text.txt efter tre körningar:

```
Stuff
Stuff
Stuff
```

Serialisering

- Det går att spara objekt på fil mellan körningar
- Java kan ”platta till” ett objekt till en sekvens av bytes
  - Perfekt för att spara
  - Använd strömmarna ObjectInputStream och ObjectOutputStream
- Kräver att klassen implementerar interfacet *Serializable*
  - Behöver dock inte implementera några metoder.

```
Object readObject(); // casting is needed to be usable
void writeObject(Object o);

ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream(new File("test.dat")));
out.writeObject(new Cat("Isaac", 11)); // if Serializable!
out.writeObject("Hejsan"); // String is Serializable
```

Serialisering

Exempel: Man vill kunna spara/ladda ett spel.

Enkel lösning: Låt spelets modell-objekt (och alla objekt som den består av) implementera *Serializable*.

Nu kan vi spara ner allt på fil!

```
GameModel gameModel = new GameModel();
                        // innehåller allt data och
                        // tillstånd för spelet.
...
// när vi vill ”spara”:
ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream(new File("savefile.xyz")));
out.writeObject(gameModel);
```

Serialisering

Exempel: Man vill kunna spara/ladda ett spel.

Enkel lösning: Låt spelets modell-objekt (och alla objekt som den består av) implementera *Serializable*.

Nu kan vi spara ner allt på fil!

```
// när vi vill ”ladda”:
ObjectInputStream in = new ObjectInputStream(
   new FileInputStream(new File("savefile.xyz")));
gameModel = (GameModel)in.readObject();
```

Undantag (eng. Exception)

Ett typiskt ohanterat undantag

Output

Server (run) | Client (run)

```
906d01d713bf8b1ef483d99ac589c372
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
        at security.AES.toStringBuilder(AES.java:165)
        at security.AES.encrypt(AES.java:91)
        at network.Client.send(Client.java:63)
        at network.Manager.ask(Manager.java:50)
        at gui.GUI.buttonFilmActionPerformed(GUI.java:579)
        at gui.GUI.access$100(GUI.java:15)
        at gui.GUI$2.actionPerformed(GUI.java:113)
        at javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:2018)
        at javax.swing.AbstractButton$Handler.actionPerformed(AbstractButton.java:2341)
        at javax.swing.DefaultButtonModel.fireActionPerformed(DefaultButtonModel.java:402)
        at javax.swing.DefaultButtonModel.setPressed(DefaultButtonModel.java:259)
        at javax.swing.plaf.basic.BasicButtonListener.mouseReleased(BasicButtonListener.java:252)
        at java.awt.Component.processMouseEvent(Component.java:6505)
        at javax.swing.JComponent.processMouseEvent(JComponent.java:3321)
        at java.awt.Component.processEvent(Component.java:6270)
        at java.awt.Container.processEvent(Container.java:2229)
        at java.awt.Component.dispatchEventImpl(Component.java:4861)
        at java.awt.Container.dispatchEventImpl(Container.java:2287)
        at java.awt.Component.dispatchEvent(Component.java:4687)
```

Detta kallas för en "stack trace"

Undantag (eng. Exception)

```
public boolean lookup(String pnr)    throws PnrNotFoundException {
  for (int i = 0;  i < last;  ++i) {
      if (contents[i].getKey().equals(key)) {
         return contents[i].getValue();
      }
   }
   throw new PnrNotFoundException("Hittade inte"  + key);
}
```

MyDictionaryImpl.java

```
...
Dictionary dictionary = new MyDictionaryImpl();
try {
  boolean passed = dictionary.lookup("880725-9505");
} catch (PnrNotFoundException e) {
   System.out.println(e.getMessage());
}
```

ex. Main.java

Undantag (eng. Exception)

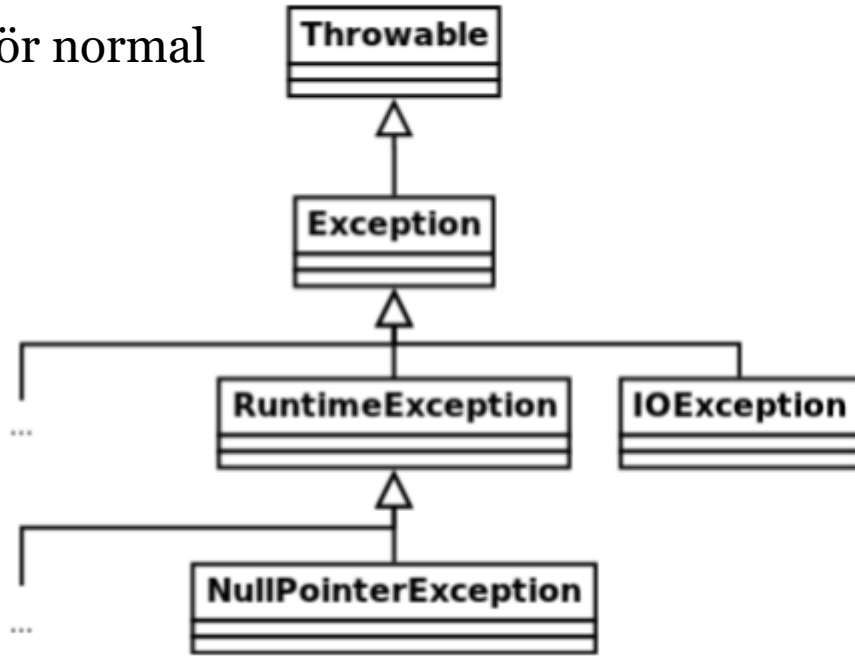
Definiera undantag

```
public class PnrNotFoundException extends Exception {
  private String pnr;

  public PnrNotFoundException(String pnr) {
    super("Could not find the person number " + pnr);
    this.pnr = pnr;
  }

  public String getPnr() {
    return pnr;
  }
}
```

Undantag (eng. Exception)

- Anger något som hänt som ligger utanför normal körning
  - Falaktig indata
  - Körfel
  - Hårdvarufel

- I Java är undantag objekt



Notera: Undantag av typen RuntimeException kräver inte throws, eller try-catch

Fundera noga på om du vill använda Exception eller RuntimeException när du skapar undantag!

Undantag (eng. Exception)

Fånga undantag

```
NESConsole myConsole = new NESConsole("MagnusNES", 1986);
try {
    myConsole.startPlaying(new NESGame("Zelda"));
} catch (NoGameCartridgeException e) {
   myConsole.ejectCartridge();
   myConsole.blowInCartridgeSlot();
   myConsole.reinsertCartridge();
} catch (ConsoleOverheatedException e) {
   myConsole.emergencyPoweroff();
} finally {
   myGame.saveAndShutdown();
}
```

Körs oavsett om undantag uppstått eller ej

Undantag (eng. Exception)

Hur man *inte* bör göra…

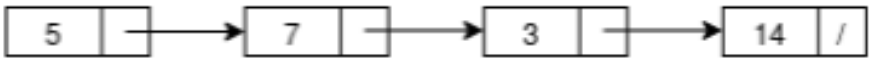
```
NESConsole myConsole = new NESConsole("MagnusNES", 1986);
try {
    myConsole.startPlaying(new NESGame("Zelda"));
} catch (Exception e) {
   // Pretend like nothing happened....
}
```

Fångar precis alla undantag!

Problem

- Vi har implementerat en länkad lista för datatypen heltal.

```
public class MyListNode {
   public int data;
   public MyListNode next;
} // Inte bra... Varför?
```



- Vi vill nu använda samma implementation igen, men den här gången för strängar.

- Hur löses uppgiften utan att kopiera all kod för varje framtida nytt behov?

Lösning?

Alla klasser ärver ju från Object…

- En lösning: Använd arv och spara alla datatyper som Object.

```
public class MyListNode {
    public Object data;
    public MyListNode next;
}
```

Men de primitiva datatyperna (int, char, double) är ju inte klasser!

Det finns motsvsarande klasser: Integer, Character, Double (s.k. ”Wrapperklasser”).

- Fungerar, men kräver explicit casting (typomvandling) varje gång ett element ska hämtas (ofta tecken på dålig objektorientering)

```
// Men tänk om datat inte är en sträng då?
String data = (String) list.remove();
```

- Leder ofta till många specialfall…

Bättre lösning

- Gör klassen generisk!
  - Mall för klasser
  - Exakta utseendet definieras först när klassen används
  - Vi kan tänka på det som en *parameter till klassen*

```
public class MyListNode<E> {
   public E data;
   public MyListNode<E> next;
}

...

MyListNode<String> node = new MyListNode<String>();
...
String text = node.data; // Ingen typomvandling behövs
```

Enligt kodkonventionerna skall dessa vara enskilda versaler. **E** står för element, **K** står för key, **V** för value o.s.v.

Generiska klasser

Fördelar:

+ Explicit casting undviks

+ Bättre möjlighet för kompilatorn att kontrollera typer
  -> Fler buggar upptäcks vid kompilering

+ Tydligare kod

+ Blir ett sätt att ”tvinga” programmeraren att göra rätt

```
ArrayList<Human> myHumanArr = new ArrayList<>();
```

Kompilatorn gissar datatypen

Generiska klasser

- Ibland uppstår behov av att begränsa vilka typer en generisk klass kan instansieras med (tydlighet, en viss sak måsta göras med datan)

```
public class MyListNode<E extends Animal> {
...}
```

- E kan här bara anta formen av Animal och dess subklasser
- Vi kan nu anropa Animal-specifika metoder på variabler av typen E

```
MyListNode<Cat> node = new MyListNode<Cat>();  // Ok

// Går ej, String är inte en subklass till Animal!
MyListNode<String> node = new MyListNode<String>();
```

Jokertecken (eng. wildcards)

- Ibland vet man inte exakt typ, men behöver ändå ange en

```
// Inte jättevackert kanske...
public void printAllAnimals(List theAnimals) {
    for (Object o : theAnimals) {
        // Är måste o typomvandlas till Animal innan vi kan anropa metoden.
        ((Animal)o).presentYourself();
    }
}
// Mycket snyggare!
public void printAllAnimals(List<? extends Animal> theAnimals) {
    for (Animal a : theAnimals) {
        a.presentYourself();
    }
}
```

Nästlade Klasser

```
public class Car {
  private int miles;
  private String plate;
  private Transmission t = new Transmission();

  private class  Transmission {
    private char[] symbols = new char[]{'N', '1', '2', '3', '4',
                                        '5', '6', 'R'};
    private char current = 'N';
    public void putIntoGear(char newGear) { ... }
  }
}
```

LINKÖPINGS UNIVERSITET

Transmission är nu en ”medlem” i Car-klassen och de kan komma åt varandras instansvariabler.

Nästlade Klasser

- Kan deklareras *public*, *private*, *protected*, eller *package private*.
- Kan deklareras static. Då får den inre klassen ej tillgång till den yttre instansen variabler.
- Icke-statiska nästlade klasser kallas också för ”inre klasser”.
- Om man bara behöver en klass inuti en metod så kan man faktiskt deklarera en klass där. Då blir det en s.k. *lokal klass*.

```
public class Car {
  private int miles;
  private String plate;

  private class Transmission {
    private char[] symbols = new char[]{'N', '1', '2',
                                        '3', '4', '5', '6', 'R'};
    private char current = 'N';

    public void putIntoGear(char newGear) { ... }
  }
}
```

Nästlade Klasser

Skuggning

- Om samma namn finns i både inre och yttre klassen så kommer endast det inre namnet finnas tillgängligt när man står där inne.
- T.ex. kommer this referera till den inre instansen.
- Man kan då använda syntaxen:
  - OuterClass.this

Nästlade Klasser

- Varför vill man göra detta?
  - Logiskt sätt att gruppera objekt som (starkt) hör ihop.
  - Ytterligare ett sätt att få inkapsling.
  - Kan leda till bättre läslighet och lättare underhåll.

```
public class Car {
  private int miles;
  private String plate;

  private class Transmission {
    private char[] symbols = new char[]{'N', '1', '2',
                                        '3', '4', '5', '6', 'R'};
    private char current = 'N';

    public void putIntoGear(char newGear) { ... }
  }
}
```

https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

Debugger

- I Eclipse kan man köra sin kod i ”debug” mode.
- Då kan man ”pausa”, ”stega” och se variablers värden

emma.enocksson@liu.se
magnus.nielsen@liu.se

www.liu.se

LINKÖPINGS UNIVERSITET