

Introduktion till objektorientering

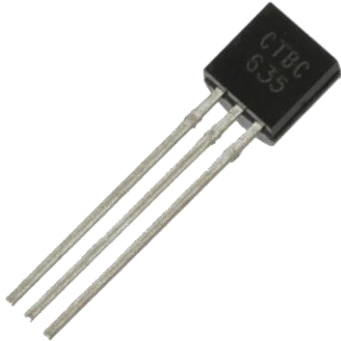
Vad är objektorientering egentligen?

Hur relaterar det till datatyper?

Hur relaterar det till verkligheten?

Perspektiv 1: Från verkligheten till objektorientering

Verkligheten innehåller mojänger

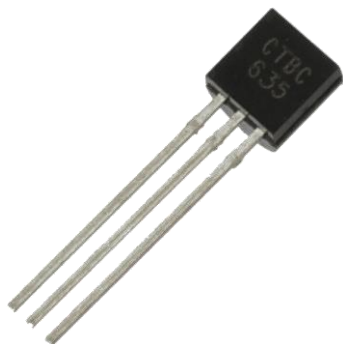


Mojängers egenskaper

Mojänger har ofta intressanta egenskaper



Längd/höjd
Färg
Motorstyrka
Toppfart



Strömtålighet
Maxfrekvens



Antal pucklar
Längd på ögonfransar

...

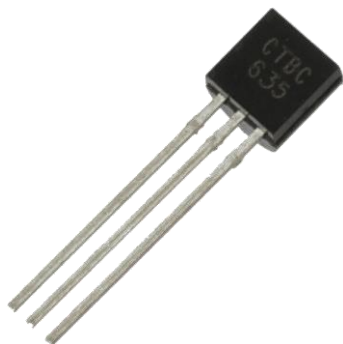
egenskaper = properties, attributes

Mojänger som gör något

En del mojänger kan **göra något**



**Accelerera/decelerera
Köra till en plats
Slå på helljuset
Stänga av vindrutetorkarna**

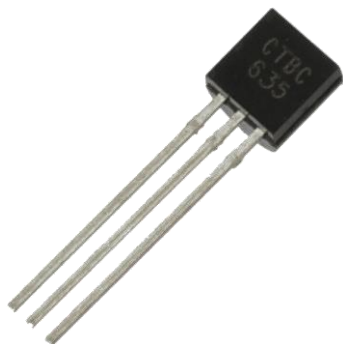


Släppa genom ström



**Gå till en plats
Äta
Spotta**

Vi behöver terminologi!



Vi har en mängd individer, **objekt**
(bilen ABC123, bilen XYZ789, ...)

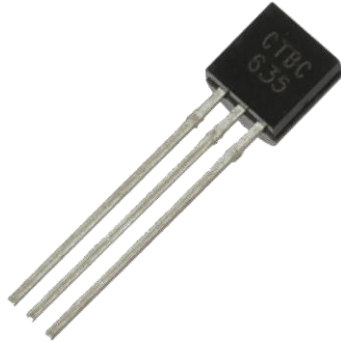
Varje **sort** av objekt är en **klass**
(klassen *bilar*, klassen *kameler*, ...)
Objekt är **instanser** av en klass

Objekt som tillhör samma klass har
samma **egenskaper** och **funktionalitet**
(*alla bilar har någon färg, kan köra*)



En **objektorienterad syn**
på den fysiska världen

När vi skriver program om/för möjänger – *autonoma bilar*:



Vi kan organisera dem som klasser!

```
// Samla all bilrelaterad kod  
// ➔ närmare koppling  
// till verklighetens objekt och klasser
```

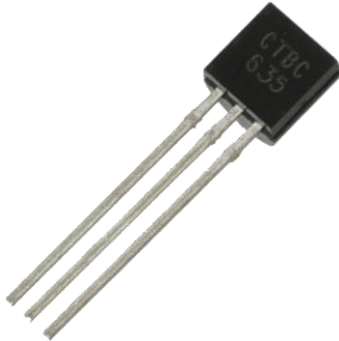
```
class Car {  
    // egenskaper  
    double length, height;  
    double topSpeed;  
    // funktionalitet  
    void driveTo(Location loc) { ... }  
    void setLights(boolean on) { ... }  
}
```

När vi skriver program om/för *mojänger* – *Googles autonoma bilar*:



Vi skapar objekt vid behov,
genom att instansiera klassen:

```
Car myCar = new Car();  
Car other = new Car();  
myCar.driveTo(home);
```



Varje bil har:

Samma beteenden (kod)

**Samma egenskaper (längd, höjd)
Egna värden (*olika* längd)**

Perspektiv 2: Från datatyper till objektorientering

(Klassbaserad objektorientering)

- Alla språk har en eller flera **primitiva** datatyper, t.ex.:

- Heltal 4711
- Flyttal 3.1415926535
- Tecken a, b, c, ...
- Sanningsvärden true, false
- ...

Primitiv:
grundläggande,
ursprunglig,
odelbar,
...

Sammanstatta datatyper: Motivering



- Nu vill vi lagra information om en **bil**
 - **Många egenskaper** för varje bil
 - *Längd – flyttal*
 - *Höjd – flyttal*
 - *Toppfart – heltal*
 - *Färgkod – heltal*
 - Ska ses som **en enhet**
 - En funktion ska ta *en bil* som *en parameter*, inte 4:
`void addToDatabase(Car c) { ... }`
 - ➔ Behöver **sammansatta** datatyper

sammansatta datatyper =
composite datatypes



- Lösning 1: Använd **generella** sammansatta datatyper

- Från Python-kursen: **Listor** till allt!

```
def skapa_bil(längd, höjd, toppfart, färg):  
    return ["bil", längd, höjd, toppfart, färg]  
minbil = skapa_bil(...)
```

- Enligt **språket** är datatypen inte **bil** utan **lista** – "*lista av vad som helst*"

```
["bil", 4.7, 1.6, ...]  
["månad", 10]  
[1, 1, 2, 3, 5, 8, 13]
```

Enligt **språket** har dessa samma
typ: Generella **listor**!

- För att kontrollera om en lista är en bil: **Uppfinn egen typkontroll!**
 - En lista **är en bil** om första elementet är strängen "bil"
 - **def** är_bil(lista): ...
 - **fordon = ["bil", "cykel", "moped", "flygplan"]...?**

- Lösning 1: Använd **generella** sammansatta datatyper

- Från Python-kursen: **Listor** till allt!

```
def skapa_bil(längd, höjd, toppfart, färg):  
    return ["bil", längd, höjd, toppfart, färg]  
minbil = skapa_bil(...)
```

- ➔ Olika värden är **indexerade numeriskt**

```
minbil[3] = 200 # Trimma, ändra toppfart
```

```
def sättMaxfart(bil, fart):  
    bil[3] = fart  
sättMaxfart(minbil, 200)
```

Numrera om
om värden tas bort
eller läggs till...

■ Lösning 2: Använd en mappning

■ Python: **dict**

```
def skapa_bil(längd, höjd, toppfart, färg):  
    return { "type": "bil",  
            "length": längd, "height": höjd,  
            "topSpeed": toppfart, "color": färg }
```

```
minbil = skapa_bil(...)
```

■ ➔ **Namngivna värden**

```
minbil["topSpeed"] = 200 # Trimma, ändra toppfart
```

- Fortfarande ingen egen typ för bilar
- Ingen standard för vilka data som finns om alla bilar
- Ingen möjlighet att kontrollera att rätt data anges / används

■ Lösning 3: Skapa nya typer med fält

■ Pascal: Poster (records)

```
Type bil = Record  
  längd, höjd, toppfart: Real;  
  färg: Color;  
End;
```

Posten har namngivna
medlemmar (fält)
(finns även i Python!)

```
Var minbil : bil;
```

Kan *deklarera* en
variabel av typ *bil*, inte
av typ *lista*
Tillåter *statisk*

typkontroll

```
minbil.toppfart := 200
```

Egenskapers värden
tas fram via *namn*,
inte via *index*

post = record
medlem = member
fält = field

■ Nästa exempel: **Datum**

- Deklarera först en **datatyp** – en "ritning", talar om *hur typen ser ut*

Exempel i "C-liknande" språk: **struct**

```
struct Date {  
    int year;  
    int month;  
    int day;  
}
```

- Skapa några **instanser** av typen – allokerar *minne*

Exempel i "C-liknande" språk

```
Date date1;  
Date date2;
```



year: 0
month: 0
day: 0

year: 0
month: 0
day: 0

Sammansatta datatyper (6)



- Fler exempel på sammansatta datatyper
 - Vanliga, "fundamentala" typer
 - `String`
 - Generella behållare
 - `List`, `Queue`, `Stack`
 - Specialiserade typer för I/O, GUI, databaser, ...
 - `InputStream`, `Window`, `Connection`, ...
 - Och typer specifika för ett visst program
 - `Driver`, `Car`, `Wheel`
 - `Shape`, `Rectangle`, `Circle`
 - `Customer`, `BankAccount`, `Transaction`
 - `Literal`, `Conjunction`, `Disjunction`, `Quantifier`

- Varje ny **datatyp** blir ett **nytt meningsfullt begrepp**!
 - Vi **samlar ihop** information – och ger den också ett **namn**
 - → Nya ord införs i vårt programmeringsspråk (**Customer**, **BankAccount**); inte fast i språkets egna begrepp (lista, dict, ...)
 - → Vi kan lättare **skriva** kod, **förstå** existerande kod:
Begreppen kan anpassas mer till **hur vi tänker**
("en kund", inte "en lista med dessa 17 kundrelaterade värden")
 - **Söndra och härska** (divide and conquer)!
 - Dela upp programmeringen i **delar av lämplig storlek**
 - Se till att varje enskild del kan **förstås i detalj**

Nu har vi samlat ihop information – men hur bearbetar vi den?

Operationer på sammansatta datatyper

- Utan OO: Vi hanterar datumen med funktioner

Exempel i "C-liknande" språk

```
boolean isDivisibleBy(int large, int small) {  
    return (large % small) == 0;  
}
```

```
boolean isLeapYear(Date date) {  
    if (isDivisibleBy(date.year, 4) && !isDivisibleBy(date.year, 100)) return true;  
    if (isDivisibleBy(date.year, 400)) return true;  
    return false;  
}
```

Vanlig syntax:

post punkt medlem
date punkt year

Vår kod...

```
add(myDate, 14);  
if (isLeapYear(myDate)) { ... }
```

anrop

Operationer på datum: Funktioner

```
boolean isLeapYear(Date date) {  
    return ((date.year % 4 == 0) && (date.year % 100 != 0) ||  
            (date.year % 400 == 0));  
}  
int daysSinceToday(date) { ... }  
void add(date, days) { ... }
```

Inspekterar och
manipulerar “utifrån”

myDate

year: 2021
month: 12
day: 31

Själva posten består
bara av passiva data!

Operationer på objekt

00 1: En utökning till poster

- Med (klassbaserad) objektorientering:
 - Post-typer blir **klasstyper** med **två sorters** medlemmar
 1. Fält eller medlemsvariabler, precis som tidigare
 2. Metoder eller medlemsfunktioner, nära kopplade till datatypen

Exempel i "C-liknande" språk

```
class Date {  
    int year;  
    int month;  
    int day;  
  
    int daysSinceToday() { ... }  
    boolean isLeapYear() { ... }  
    void add(days) { ... }  
}
```

Vad "vet" ett datum?
Vilken information finns?

Vad kan ett datum
göra?

klass = class
fält = field
medlemsvariabel = member variable
metod = method
medlemsfunktion = member function

00 2: Be objektet att göra något

- Själva värdet kallas inte **post** utan **objekt**



Vår kod...

```
Date pastDate = ...;  
pastDate.add(14);  
if (pastDate.isLeapYear()) ...;
```

Vanlig syntax:
objekt punkt medlem

Vi kan **be datumet tala om för oss**:
"Infaller **du** under ett skottår?"

- **Fundamental** princip:
"Klassen bestämmer över sina objekt"
(kan förhindra manipulation utifrån)
- **Grunden** för
ärvning, overriding, polymorfism, inkapsling, ...

- Varje klass blir ett ”djupare” begrepp!
 - ”Detta är en sträng, och alltså kan den...”:
Helhetsförståelse för vad ett objekt vet och vad det *kan göra*
 - Söndra och härska – mellan olika datatyper
 - Men *samlar ihop kod och information* inom en och samma datatyp

UML: Unified Modeling Language

En mycket kort introduktion

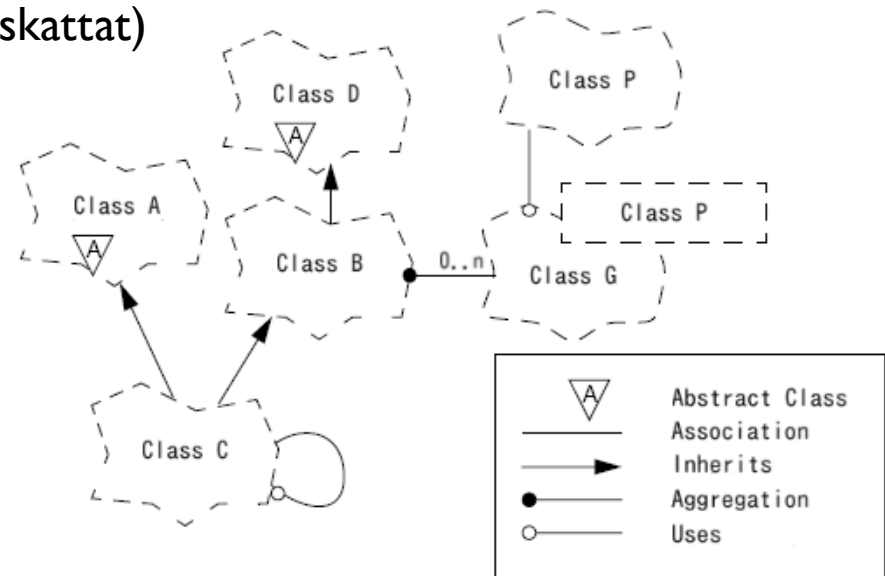
■ Program kan vara **stora**

■ Rader kod enligt <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>

- Unix 1.0: 10 000
- Photoshop 1.0: 120 000
- Windows 3.1: 2 500 000
- Open Office: 23 000 000
- Firefox: 36,890,150 (~~9 700 000~~)
- Typisk bil, 2013 100 000 000
- Google, 2015 2 000 000 000 (uppskattat)

■ Hur håller man koll på detta?

- **Metoder** för modellering
- **Visualisering** av modeller –
kräver standardiserad *notation*

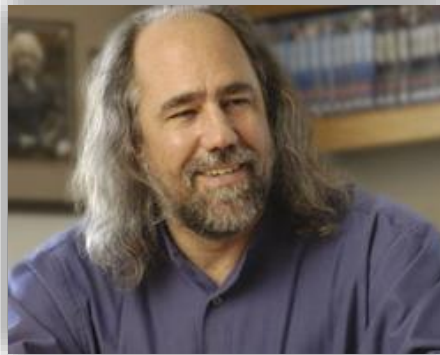


- Tidigt 90-tal: Tre vanliga metoder och notationer för OO-modeller

**James Rumbaugh:
Object-modeling
technique (OMT)**



**Grady Booch:
Booch-metoden**

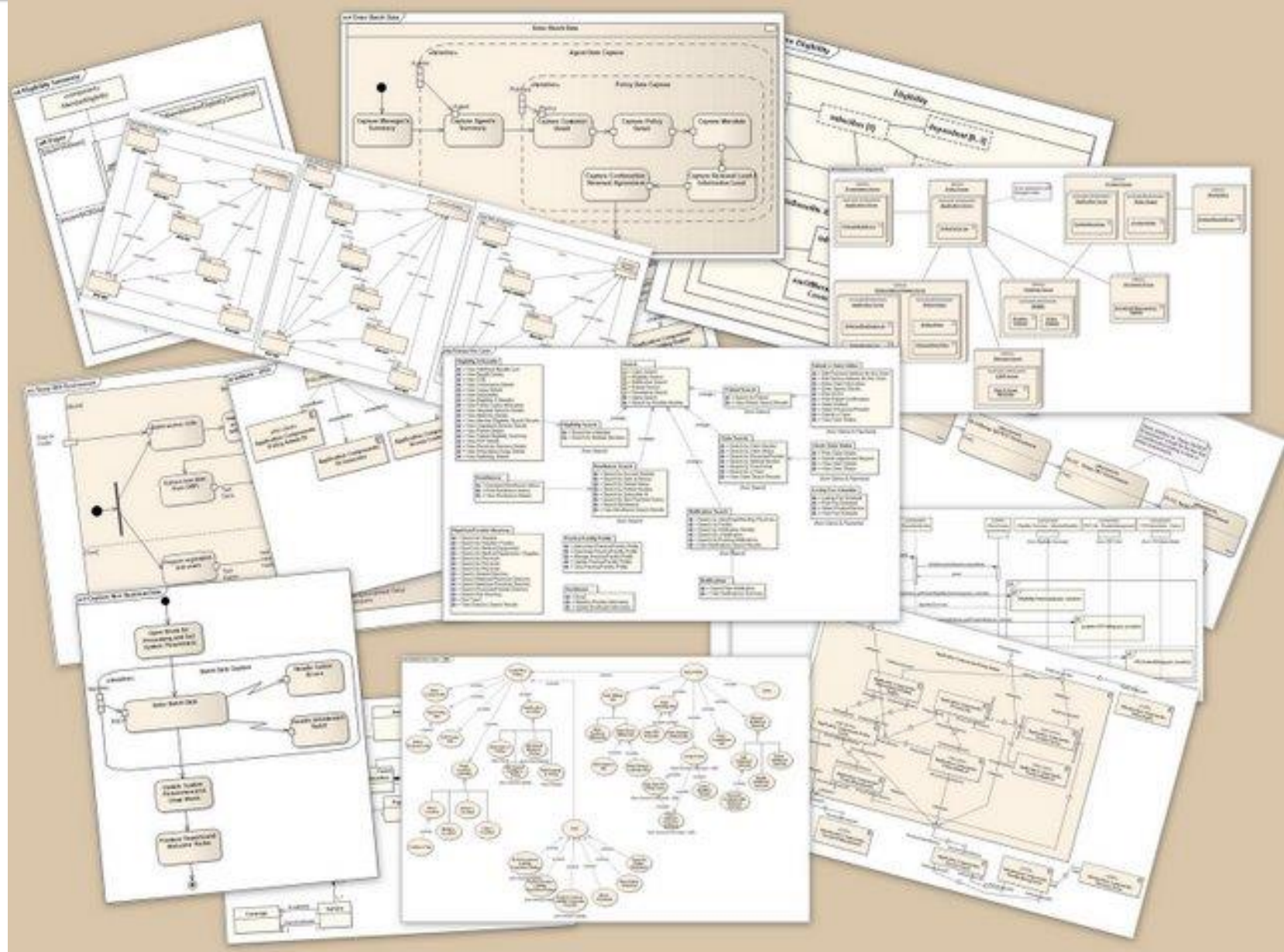


**Ivar Jacobson:
Object-Oriented
Software Engineering
(OOSE)**

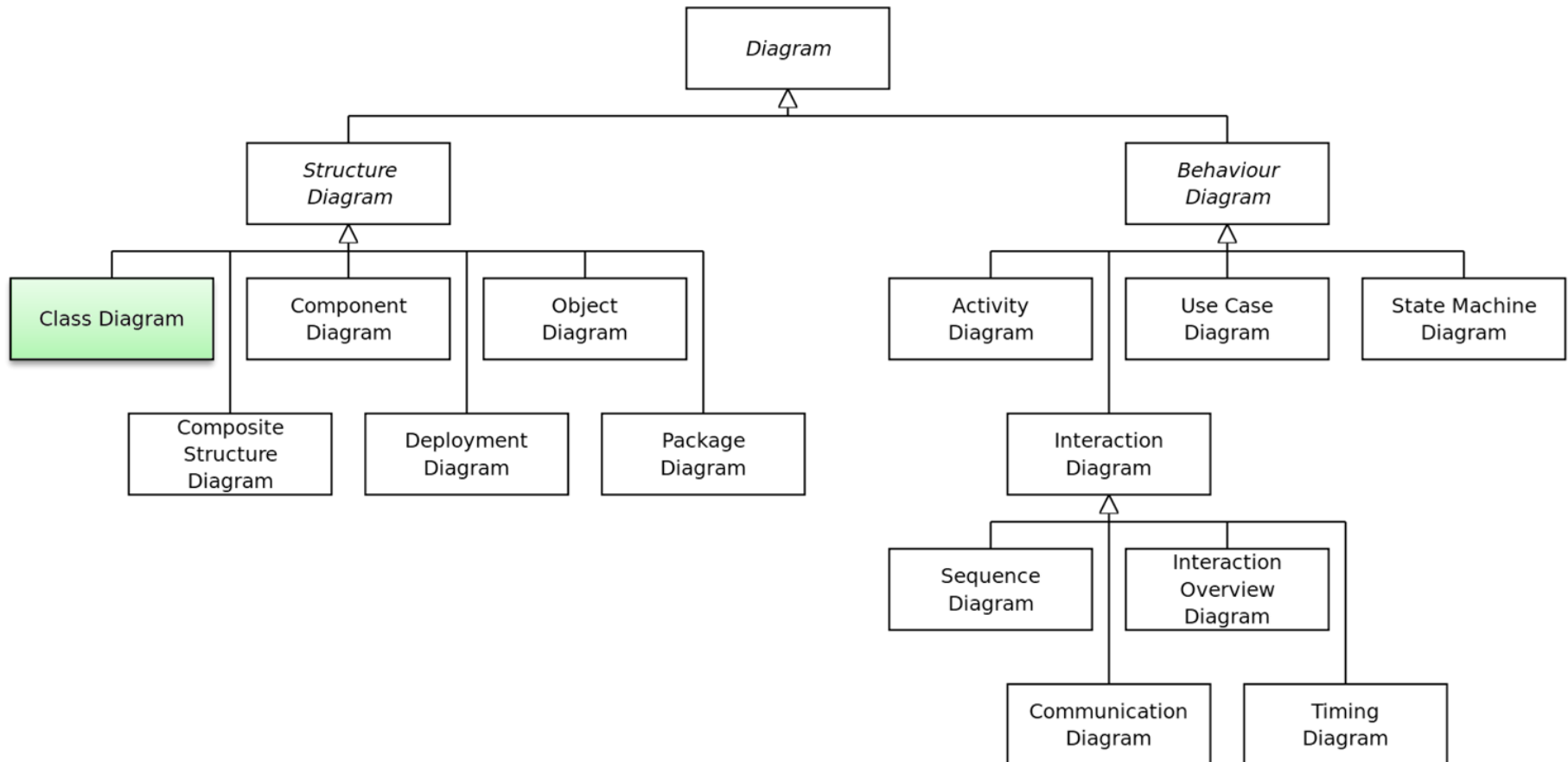


- 1995: Anställda på samma företag, kända som The Three Amigos
 - 1996: För många metoder → långsammare spridning av OO
 - Ledde konsortium som skapade Unified Modeling Language, UML
 - Ingen utvecklingsmetod – ett (delvis visuellt) språk

UML-diagram



- Vi fokuserar på en enkel form av klassdiagram!



- Strukturen hos en enskild klass:

ArrayList
elements: Object[] length: int
size() : int get(int amount): Object add(Object element): Object set(int index, Object element): void

Klassnamn
Attribut
Operationer (metoder)

- Vi återkommer med utökade klassdiagram
 - Relationer mellan instanser
 - Relationer mellan klasser