# Lösningsförslag till tentamen för TDA551
# Objekt-orienterad Programmering och Design

*Dag:* 2017-01-10, *Tid:* 14.00-18.00

## Uppgift 1

a) En statisk metod tillhör en viss klass och inte ett objekt. En statisk metod kan bara läsa statiska variabler och inte påverka tillståndet av objekt, dvs modiefiera instansvariabler.

b) En dynamisk typ används vid en dynamisk typkontroll som sker vid exekvering. Dynamiska typen för en variabel är typen av objektet som variabeln referar till.

c) Ett undantag (exception) är ett objekt som påtalar en ovanlig eller felaktig situation i ett program. Undantag kastas av programmet och kan fångas och hanteras.

d) En typ-parameter är en parameter till en generisk klass eller metod. En generisk klass (eller metod) är generaliserad över en (eller flera) typ.

## Uppgift 2

a) Inkorrekt och ger en statiskt fel: `AircraftFighter` är abstrakt och man kan inte skapa objekt av abstrakta klasser.

b) Inkorrekt och ger en statiskt typ fel: det finns ingen konstruktor med inga parametrar.

c) Korrekt: först blir `public void shoot()` i `JetWithTurbo` klassen anropad och sedan `public void shoot()` i `JetFighter` klassen.

d) Korrekt: först blir `public void shoot()` i `JetWithTurbo` klassen anropad och sedan `public void shoot()` i `JetWithGun` klassen.

## Uppgift 3

a) i) Korrekt.

ii) Inkorrekt: `JetFighter jf = listExtendsAF.get(0)` ger ett typfel. Vi kan bara läsa element med typen `AircraftFighter` (eller dess super typer).

iii) Inkorrekt: `listAF = listJF` ger ett typfel. En generisk lista är invariant i sin typ-parameter.

iv) Inkorrekt: `AircraftFighter af = listSuperJF.get(0)` ger ett typfel. Vi kan bara läsa element av typen `Object` för `listSuperJF` är contravariant i sin typ-parameter.

b) `listAF`

c) Vad `?` än är så vet vi att vi att den ärver från `T` och därför kan vi läsa från listan men inte skriva. Till exempel om `T` är `AircraftFighter` då kan vi ta emot listor med elementtyp `JetFighter` eller `JetWithTurbo` för de är subtyper till `AircraftFighter`, så vi kan anropa metoder från `AircraftFighter` utan problem. Lägga till saker går inte, vi kan ju inte lägga till en `JetWithTurbo` i en lista av `JetFighter`.

## Uppgift 4

a) En klass är trådsäker om den garanterar konsistens för sina objekt, även om det finns flera trådar närvarande. Vi kan få problem med 'race conditions' om en klass är inte trådsäker.

b) Nej, metoden `fireMissile` borde vara `synchronized`, man kan hamna i en ogiltig situation.

## Uppgift 5

a) Nej, klassen är icke-muterbar. För att vara muterbart borde klassen och instansvariabler vara icke-`final` och antigen publika eller ha 'mutators'.

b)
```
public final class Point {
  private final int x;
  private final int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public int getX() {
    return x;
  }

  public int getY() {
    return y;
  }

  public Point setX(int x) {
    return new Point(x, y);
```

```
  }

  public Point setY(int y) {
    return new Point(x, y);
  }

  public Point move(int x, int y) {
    return new Point(x, y);
  }
}
```

## Uppgift 6

a) Single Responsibility principen säger att en klass ska ha bara en anledning för att ändra. T.ex. anta att vi har ett program för att analysera textfiler och det finns en klass som både översätter texten till en intern representation och gör analysen:

```
class TextAnalyzer {
  public void parseString(String text) {};
  public boolean analyze() {};
  ...
}
```

Då bryter vi mot SRP och är det bättre dela upp klassen.

```
class TextAnalyzer {
  public boolean analyze(Text text) {};
  ...
}

class Parser {
  public Text parseString(String text) {};
  ...
}
```

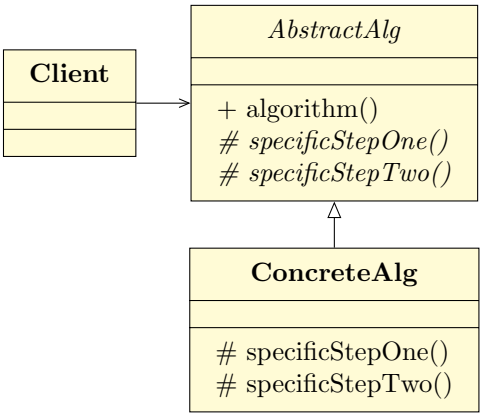
b) Law of Demeter principen säger att vi ska undvika att introducera beroenden genom att hålla oss till följande riktlinjer:

- En klass ska bara känna till sina närmaste "vänner" (dvs ofrånkomliga beroenden).
- En klass ska inte anropa metoder hos andra än sina vänner.

T.ex. antar att vi har en `Point` instansvariabel, då bryter man mot principen om man anropar `move` metoden så här `obj.getPosition().move(0, 42)`. I en sådan fall är det bättre att introducera en egen `move` metod.

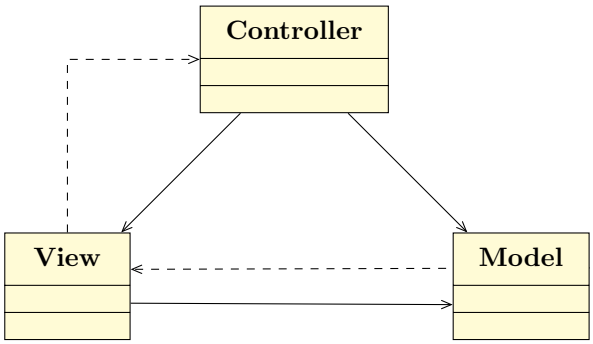
## Uppgift 7

a) Template method design mönstret definiera ett skelett av en algoritm och vidarebefordrar några steg till konkreta subklasser. Om man har flera algoritmer som har mycket gemensamt men bara skiljer sig på några detaljer, då kan man implementerar gemensamma delen i en abstrakt klass och använder abstrakta metoder för de specifika delarna.



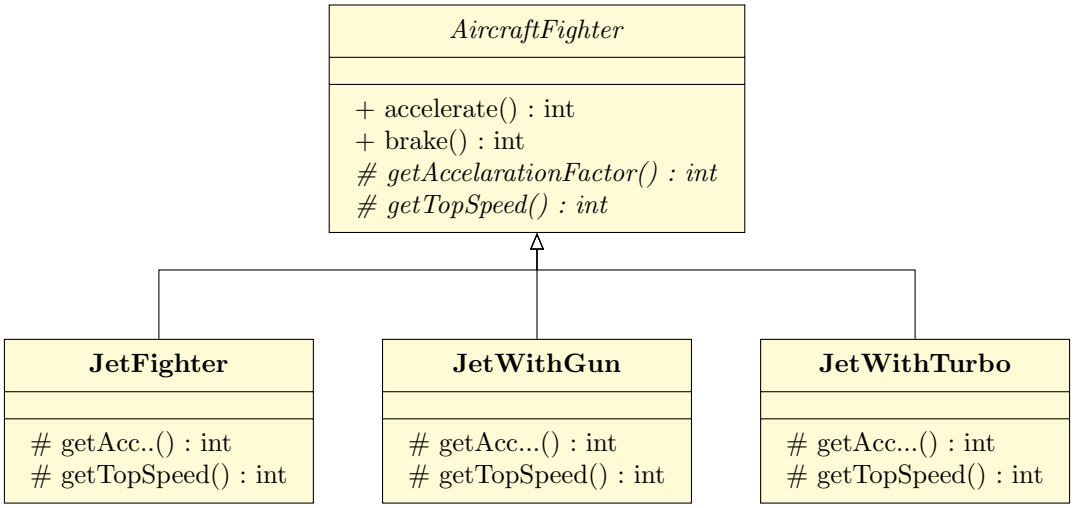

b) A model-view-controller design mönster delar är ett architectural mönster och delar upp ett program i en model som tar hand om datan och logiken, en view som bestämmer hur vi representerar datan, och en controller som agerar på handelser utifrån. Syftet med mönstret är att vi separerar datamodellen från användaregränssnittet.

Vi vill ha en SMART model", där all domänlogik ligger, och ingenting saknas. Vi vill ha en DUMB view", som enbart sköter presentationen.Vi vill ha en THIN controller", som lyssnar på användar-input och hanterar dessa genom att skicka rätt meddelanden till modellen och vyn.



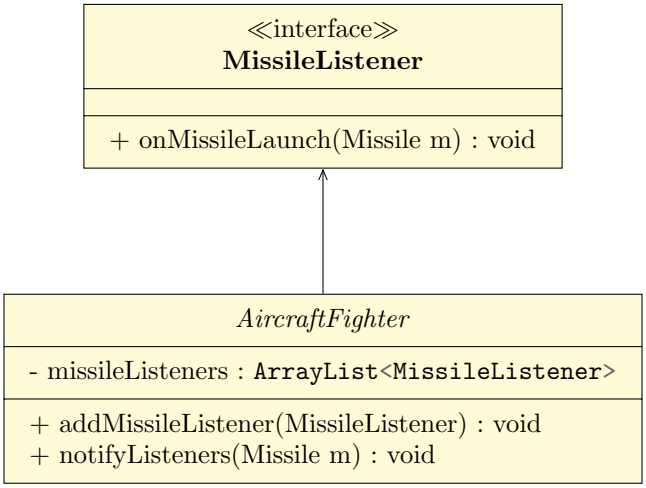
# Uppgift 8

## Template Method Pattern

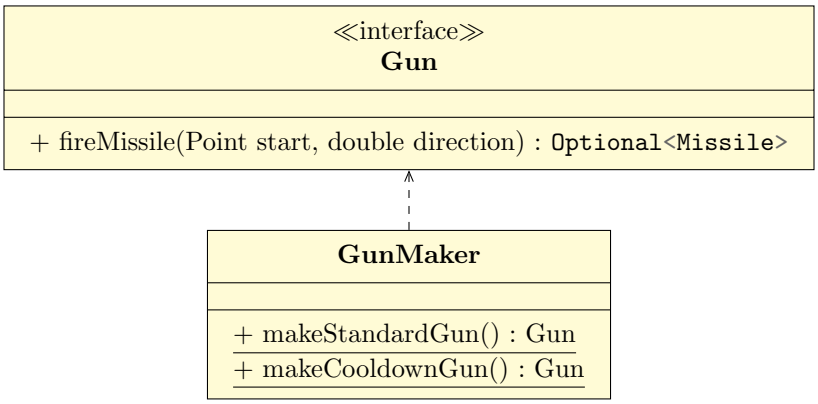



## Observer Pattern

Vi har inte någon konkret observer men det spelar ingen roll, kärnan i Observer Pattern är att möjliggöra för sådana att existera.

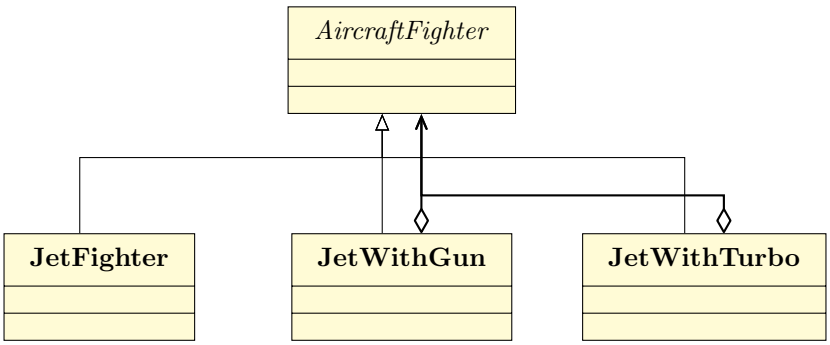


### Factory




### Decorator

`JetFighter` är basfallet med `JetWithTurbo` och `JetWithGun` som 'decorators', som dynamiskt tillför extra funktionalitet.




### Chain of Responsibility

Många fall, e.g. `JetWithGun.turnLeft` delegerar anropet vidare till sin inre `aircraft`, tills en konkret handler hittas.

### Missförstånd

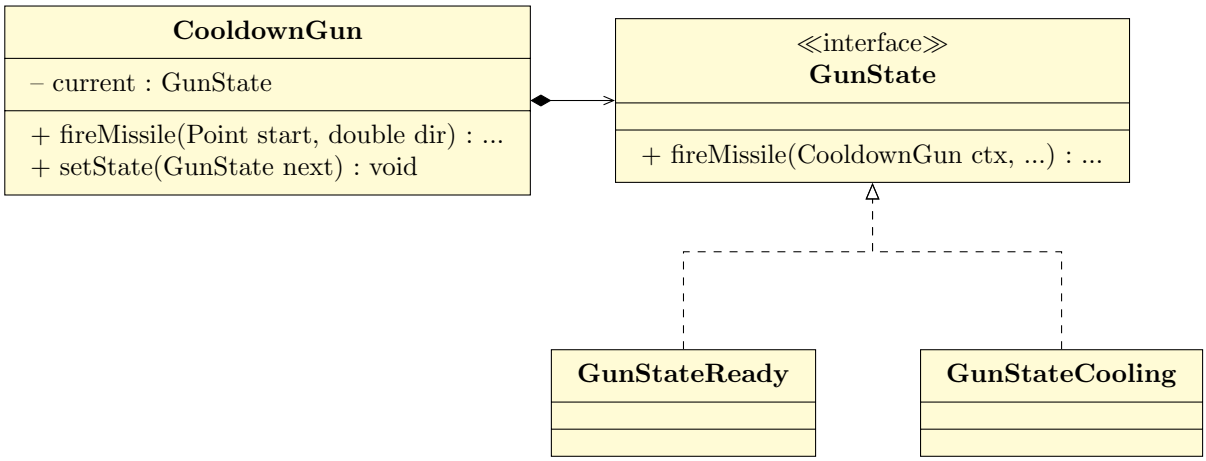
De vanligaste två missförstånden:

- Template Method Pattern talar om en template method - dvs en kontext-oberoende metod i superklassen som beror av kontext-beroende metoder som är abstrakta i superklassen och implementeras konkret i subklasserna. E.g. `AircraftFighter.shoot` är *inte* en template method - den är helt kontext-beroende (abstrakt), och är ett exempel på helt vanlig subtypnings-polymorfism. Det som gör TMP speciellt är just kombinationen mellan kontext-oberoende (code reuse) och kontext-beroende (polymorfism).

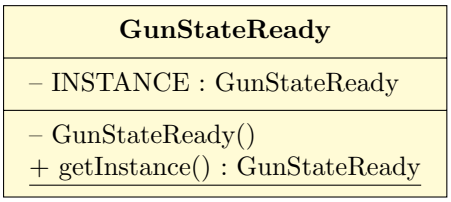
- Relationen mellan `JetWithGun` och `Gun` är inte State Pattern - då hade vi behövt ett sätt att växla mellan olika guns. Det är inte heller ett Strategy Pattern - då hade vi förväntat oss en metod som tog en strategy (dvs `Gun`) som argument, och agerade därefter.

## Uppgift 9

a) Se koden lite längre ner.




b) Här bad jag enbart om UML, så till exempel för `GunStateReady`:




### Koden

Nedanstående kod visar en lösning för hela uppgiften.

### CooldownGun

```
import java.util.Optional;

public class CooldownGun implements Gun {
  private GunState current = GunStateReady.getInstance();
  private int cooldownLength;

  public CooldownGun(int cooldownLength) {
```

```
    this.cooldownLength = cooldownLength;
  }

  public void setState(GunState next) {
    current = next;
  }

  @Override
  public Optional<Missile> fireMissile(Point start, double direction) {
    return current.fireMissile(this, start, direction);
  }

  public void startCooldown() {
    new Cooldown().start();
  }

  class Cooldown extends Thread {
    @Override
    public void run() {
      try {
        Thread.sleep(cooldownLength);
      } catch (InterruptedException e) {
        // if interrupted, just proceed with setting variable to false.
      } finally {
        current = GunStateReady.getInstance();
      }
    }
  }
}
```

---

### GunState

```
import java.util.Optional;

public interface GunState {
  Optional<Missile> fireMissile(CooldownGun ctx, Point start, double direction);
}
```

---

### GunStateReady

```
import java.util.Optional;

public class GunStateReady implements GunState {
  private final static GunStateReady INSTANCE = new GunStateReady();

  private GunStateReady() {};
```

```
  public static GunStateReady getInstance() {
    return INSTANCE;
  }

  @Override
  public Optional<Missile> fireMissile(CooldownGun ctx, Point start, double direction) {
    ctx.setState(GunStateCooling.getInstance());
    ctx.startCooldown();
    return Optional.of(new Missile(120, direction, 100));
  }
}
```

---

### GunStateCooling

```
import java.util.Optional;

public class GunStateCooling implements GunState {
  private static final GunStateCooling INSTANCE = new GunStateCooling();

  private GunStateCooling() {
  }

  public static GunStateCooling getInstance() {
    return INSTANCE;
  }

  @Override
  public Optional<Missile> fireMissile(CooldownGun ctx, Point start, double direction) {
    return Optional.empty();
  }
}
```

## Uppgift 10

a)
- `AircraftFighter` rad 43 (beroende på `ArrayList` isf `List`)

b)
- `AircraftFighter` rad 9, 14, 26, 31 (metoderna ändrar tillstånd och returnerar ett värde)