Designprinciper - OOP

SOLID

SRP-**O**CP-**L**SP-**I**SP-**D**IP

Single Responsibility Principle (SRP)

> A class should have at most one reason to change.

- Vad fyller principen för syfte?
  - Förändringar hålls väl avgränsade, så om funktionalitet behöver ändras minskas risken för sammanblandning (robust).
  - Detta gör även klassen lättare att testa separat från annan funktionalitet, samt att all kod som rör ett visst ansvarsområde finns på ett enda ställe (easy access).

- Varför är det bra att följa den?
  - Det är enklare att förstå, testa och felsöka en klass som har ett väl avgränsat ansvarsområde.
  - Dessutom kan man ändra i klassen utan att oroa sig för att det påverkar andra delar.

Open-Closed Principle (OCP)

> Software modules should be open for extension, but closed for modification.

- Vad fyller principen för syfte?
  - Att skriva kod där ny funktionalitet enkelt kan läggas till (open for extension) samtidigt som så mycket som möjligt av existerande kod kan lämnas orörd (closed for modification).

- Varför är det bra att följa den?
  - Koden blir maintainable och extendable.

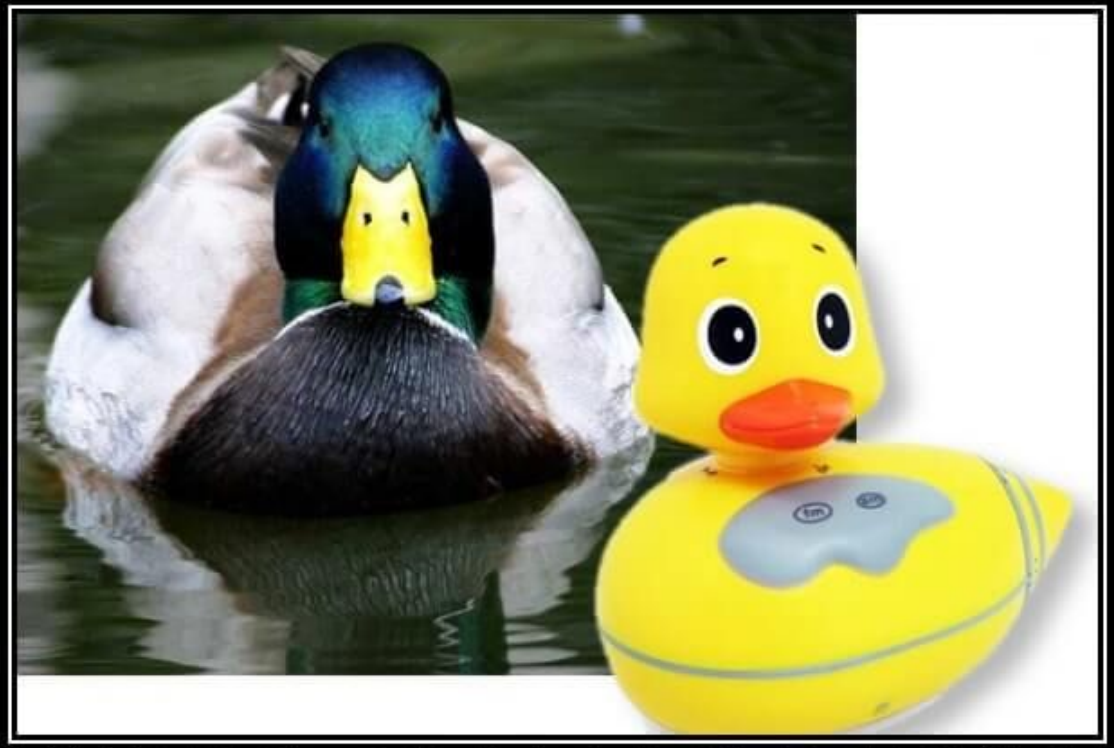
Liskov Substitution Principle (LSP)

Beskriver när arv bör och inte bör användas.

> If for each object **o1** of type **S** there is an object **o2** of type **T** such that for all programs P defined in terms of **T**, the behavior of P is unchanged when **o1** is substituted for **o2**, then **S** is a subtype of **T**. (*Barbara Liskov*)

- Vad fyller principen för syfte?
  - Den förtydligar hur en arvstruktur ska se ut (IS-A förhållande), samt sätter upp riktlinjer för när det är lämpligt att faktiskt använda sig av arv. Exempelvis ska subklasser lägga till extra beteenden / beräkningar, de ska inte ändra befintliga beteenden / beräkningar från superklassen.

- Varför är det bra att följa den?
  - Det blir en tydlig struktur vilket gör koden mer lättbegriplig samtidigt som man slipper oväntade beteenden som annars kan uppstå



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Interface Segregation Principle (ISP)

> No client should be forced to depend on methods it does not use.

- Vad fyller principen för syfte?
  - Genom att bryta ut större interfaces i flera mindre, behöver klienter inte implementera dummy-metoder som de inte ens använder. Fungerar som SRP, fast för interfaces.

- Varför är det bra att följa den?
  - Man slipper onödig kod och man slipper även skriva om koden när en metod, som man inte ens använder, byter signatur.

Dependency Inversion Principle (DIP)

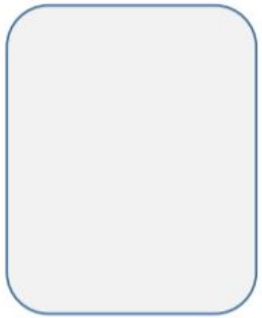
Moduler???

Modulär design *(Architectural design pattern)*

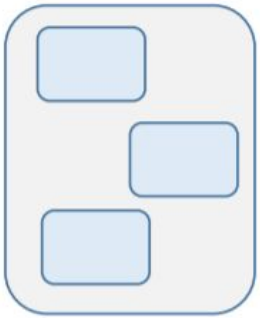
- Ett programsystem är för stort för att kunna förstås i sin helhet. Vi måste bryta ner det i delsystem för att förstå. Denna process kallas dekomposition.

- Ett modulärt system är uppdelat i identifierbara abstraktioner på olika nivåer – t ex classes, delkomponenter, paket, subsystem.

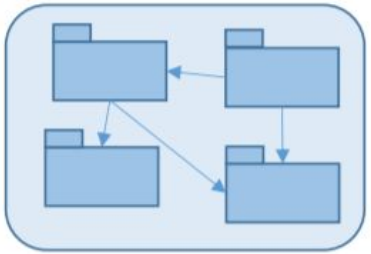
- Fördelar med en välgjord modulär design:
  - Lätt att utvidga
  - Moduler går att återanvända
  - Uppdelning av ansvar
  - Komplexiteten reduceras
  - Moduler går att byta ut
  - Tillåter parallell utveckling



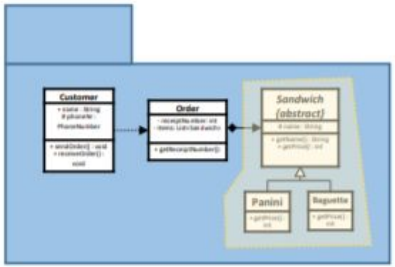
System



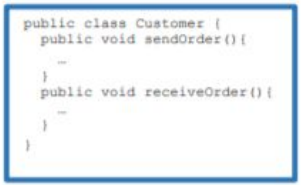
Subsystem



Packages



Delkomponenter
Classes



Metoder

Dependency Inversion Principle (DIP)

> Depend on abstraction, not on concrete implementations.

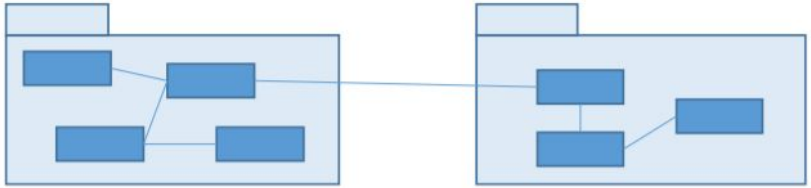
- Vad fyller principen för syfte?
  - High-level modules should not depend on low-level modules.
    Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions. Vi vill skapa så få och lösa beroenden som möjligt, samt att dessa beroenden är avsiktliga. Genom att använda supertyper istället för subtyper kan vi minska beroendet av en specifik klass.

- Varför är det bra att följa den?
  - Vi får kod med låg coupling och mer flexibilitet

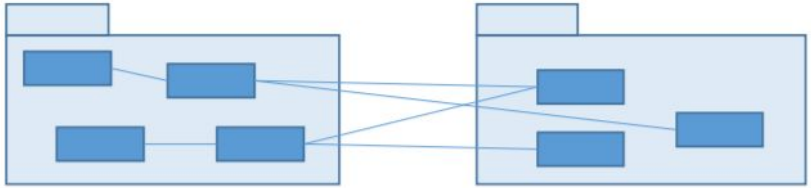
Generella principer

Cohesion

- Cohesion (sammanhållning) är ett mått på den inre sammanhållningen i en modul.
- Hur väl samverkar komponenterna inom modulen?
- Vi eftersträvar high cohesion – dvs när samtliga komponenter samverkar för att lösa modulens ansvarsområde, utan att behöva samverka med komponenter i andra moduler.
- Hur autonom är modulen? Klarar den sitt uppdrag på egen hand?
- Hur väldefinierat är modulens ansvarsområde?



High cohesion

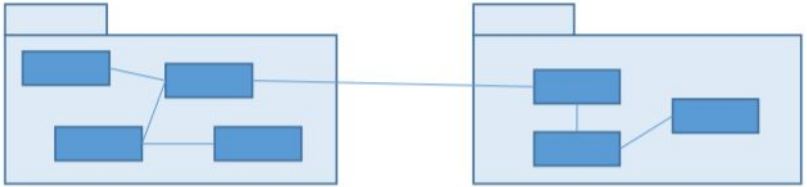


Low cohesion

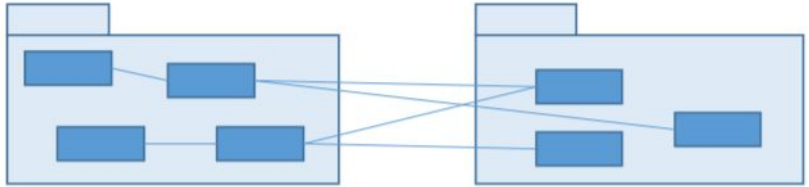
Coupling

- Coupling (sammanbindning) är ett mått på hur starkt beroendet är mellan två olika moduler.
- Hur autonom är modulen? Klarar den sitt uppdrag på egen hand?
- Hur väl avgränsad är modulen? Tillåter den andra moduler att bero på dess inre implementation?
- För att få en flexibel och modulär design måste ingående moduler vara så oberoende av varandra som möjligt. Vi eftersträvar low coupling mellan moduler.

- Har vi starka kopplingar kommer förändringar i en modul framtvinga förändringar i andra moduler som är beroende av den – bryter mot OCP.



Low coupling



High coupling

Command-Query Separation

Asking a question should not change the answer.

En metod ska antingen ha sidoeffekter eller returnera ett värde, inte båda.

- Vad fyller principen för syfte?
  - Man ska kunna anropa en query-metod (metod med returtyp) utan att oroa sig för att programmets tillstånd ändras.
- Varför är det bra att följa den?
  - Man slipper bli överraskad av oväntade beteenden.

Composition Over Inheritance

Favor composition over inheritance.

- Vad fyller principen för syfte?
  - Den ger lösare beroenden och mer robust kod jämfört med arv (där subklassen helt kan sluta fungera om det sker en förändring i superklassen).
  - Koden blir även mer flexibel då Java inte stödjer multipla implementationsarv, och det är även möjligt att återanvända kod där arv inte är lämpligt.
- Varför är det bra att följa den?
  - Läs ovan

Law of Demeter

Don’t talk to strangers

- Vad fyller principen för syfte?
  - Vi undviker att introducera beroende. En klass bara ska känna till sina närmsta "vänner" och klassen ska inte heller anropa metoder hos andra än sina vänner.
- Varför är det bra att följa den?
  - Åstadkommer kod med låg coupling, är testbar och som kan återanvändas.

Separation of Concern (SoC)

Som SRP fast på högre nivå.

- Vad fyller principen för syfte?
  - Att separera ett program i distinkta sektioner så att varje sektion hanterar sitt eget ansvarsområde / logik. Detta kan göras på hög och låg nivå.
  - På hög nivå kan vi bryta ut ett program i sektioner som UI, företagslogik, användarlogik och databasen. MVC är ett bra exempel på SoC där man delar upp programmet i en Modell, View och Controller. På låg nivå fungerar SoC som SRP, fast SoC hanterar mer än bara klassen.

- Varför är det bra att följa den?
  - Då kan flera programmerare jobba på olika sektioner isolerat från varandra, koden blir reusable, enklare att underhålla och testa.

Design patterns

Mer om det nästa lektion