

Undantag, Sammanfattning och Tentamensinfo

Objektorienterad programmering och design

Alex Gerdes, 2017

Saker går fel

- ... av många olika anledningar.
 - Kass kod. Programmeraren har skrivit kod med buggar, som e.g. refererar till ett objekt som inte finns, indexerar utanför en array, dividerar med 0, etc.
 - Felaktig användning av API. Programmeraren har inte läst dokumentationen, och uppfyller inte uppställda villkor, e.g. skickar ett negativt tal till en metod som bara hanterar positiva tal.
 - Externa faktorer som programmet inte har kontroll över, e.g. nätverket går ner, databasen kraschar, DVD:n är inte isatt, etc.

Felhantering

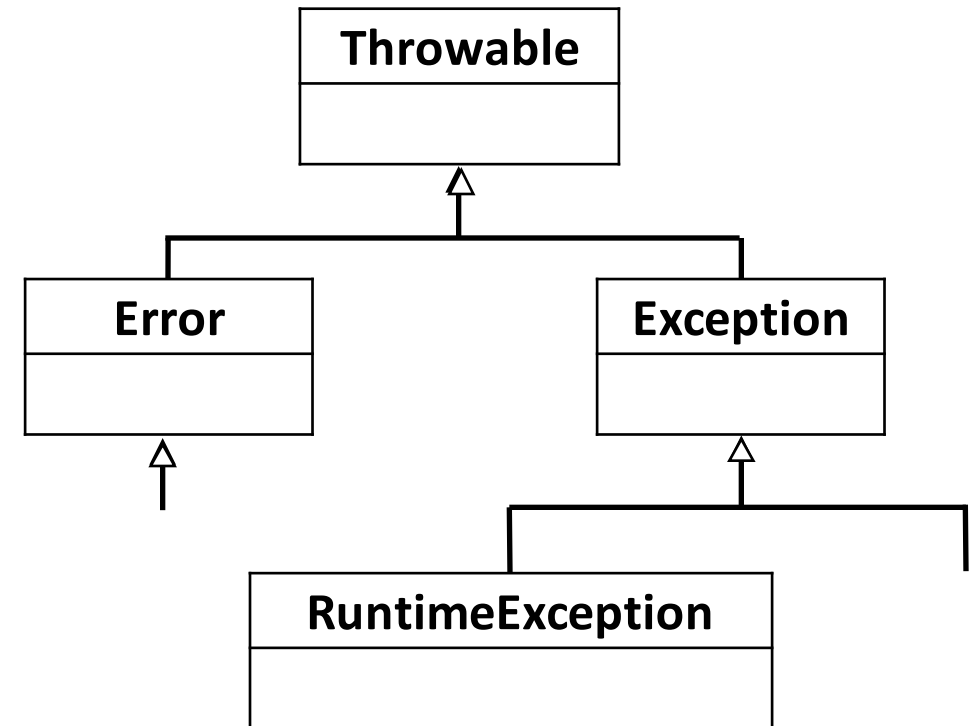
- Äldre programspråk (som C) har ingen strukturerad felhantering. Fel måste signaleras till användaren via värdet som returneras, så kallade *felkoder*.
 - Leder till olika konventioner, kraftigt ökad komplexitet, etc.
- Moderna imperativa programspråk (som Java) har strukturerad felhantering via *exceptions*. Detta innebär att vi slipper använda felkoder.
 - Använd *aldrig* felkoder i Java. Aldrig.
- Många använder felkoder fortfarande... don't!

Exceptions

- Ett *exception* (undantag) i Java är ett objekt som representerar, och innehåller information om, ett fel som uppstått av en eller annan anledning.
 - Ett exception kan *kastas* (`throw`).
 - Ett exception kan *fångas* (`catch`).
- När ett exception inträffar innebär det en form av *non-local transfer of control*. Koden följer inte den normala strukturen, utan kan "hoppa" till ett catch-block långt bort från där exception kastas.

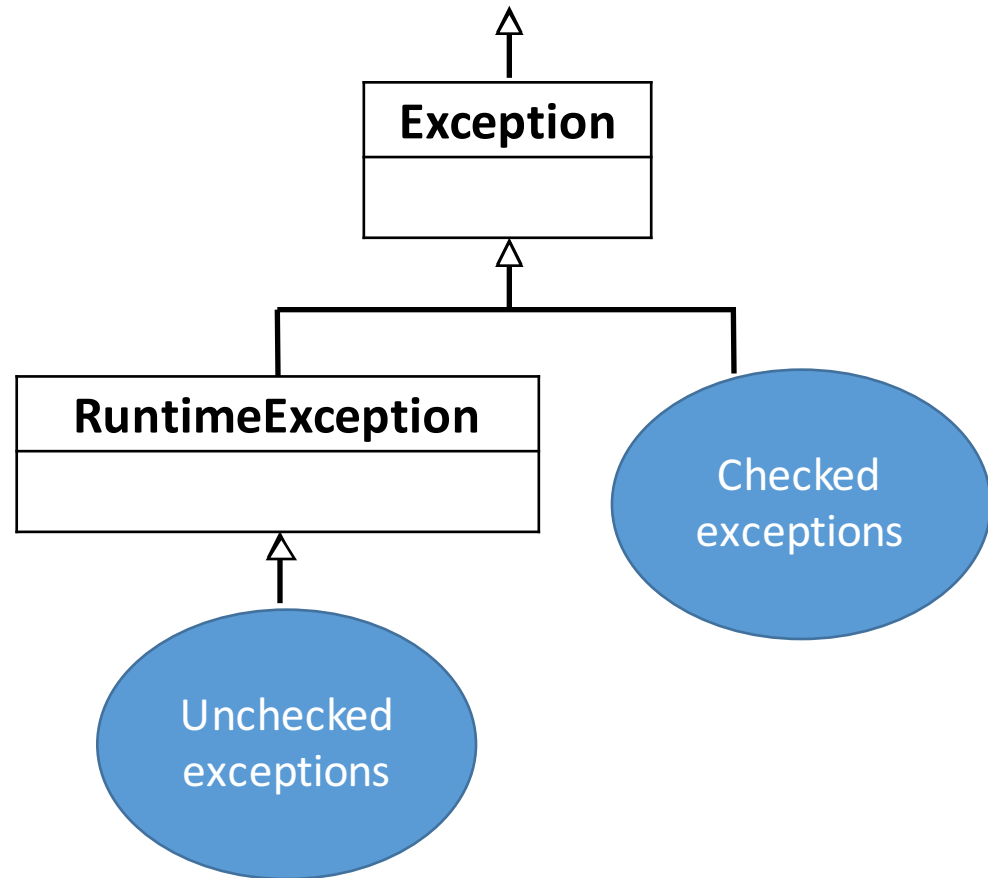
Error vs Exception

- Alla former av "fel-objekt" är i Java sub-klasser till klassen `Throwable`, som namnet till trots är en klass och inte ett interface.
- `Error` representerar ett fel som inte går att återhämta sig från, exekveringen ska avslutas.
 - E.g. `VirtualMachineError`
 - Kan fångas, men bör bara fångas för att avsluta programmet på ett " snyggt" sätt.



Checked vs Unchecked

- `RuntimeException` representerar "buggar" – saker som inte borde inträffa och därför inte "borde" behöva varnas för.
 - E.g. `ArrayIndexOutOfBoundsException`, `IllegalArgumentException`
 - Dessa är *unchecked*, dvs behöver inte deklareraras från metoder.
 - Även `Error` och dess subklasser är unchecked.
- Alla andra exceptions är *checked* – de representerar saker som vi förväntar oss kommer att inträffa under normal körning – undantagsfall, förvisso, men ändå. Dessa måste vi varna användare för.
 - E.g. `FileNotFoundException`, `SQLException`



Checked exceptions

- För checked exceptions måste vi deklarera, i metoders signaturer, om de *kan* komma att kasta exceptions av typen i fråga:

```
public String readFile(String fileName)
    throws FileNotFoundException {
    ...
}
```

- En metod som anropar `readFile` måste antingen fånga `FileNotFoundException`, eller själv deklarera att den kan komma att kasta samma exception.
 - Kallas *exception propagation*

Att fånga exceptions

```
public void appendFile(String fileName, String str) throws IOException {  
    try {  
        String contents = readFile(fileName);  
        contents += str;  
        writeFile(fileName, contents);  
    } catch (FileNotFoundException e) {  
        createFile(fileName);  
        writeFile(fileName, str);  
    }  
}
```

Vi fångar en sorts fel

Men kan fortfarande orsaka andra
sorters IOException, e.g. om vi inte
har permission att skriva.

Initialisering av objekt

Initialisering av objekt

- När vi ber om att skapa ett nytt objekt med ett anrop till en konstruktor, sätter vi igång en kedja av händelser, som (förhoppningsvis, om inga exceptions händer) mynnar ut i att vi får tillbaka ett objekt av typen i fråga.

Initialisering av objekt

1. Statisk initialisering av klassen ("maskinen startar upp").
 - `static` initializer blocks, samt initialisering för `static` attribut.
 - Görs bara om maskinen inte redan startats av ett tidigare anrop, till konstruktor eller någon `static` metod (eller användning av `static` attribut).
2. Anrop till konstruktor för objektets superklass ("maskinen utgår från tidigare modell")
 - Explicit anrop till någon `super(...)`-konstruktor måste göras allra först i en konstruktor.
 - Om ingen super-konstruktor anropas explicit, anropas implicit `super()`.
3. Initialisering av objektet ("maskinen skapar grunden")
 - Non-static initializer blocks, samt initialisering för non-static attribut.
4. Exekvering av konstruktorn självt ("maskinen färdigställer")
 - Kod som explicit skrivits i konstruktor, *förutom* eventuellt anrop till en super-konstruktor.

Exempel på initialisering

```
public class Init {  
    static String hello = "Hello!";  
    int x = 5;  
    String foo;  
  
    Init() {  
        constructorCode();  
    }  
}
```

```
Init init = new Init();  
=====  
hello = "Hello!";  
  
super();  
  
x = 5;  
foo = null;  
  
constructorCode();
```

Ett anrop av
konstruktorn
resulterar i...

1. static init

2. super

3. object init

4. constructor

Default constructor

- Alla klasser har en *default constructor* som skapas oavsett om den skrivs ut eller inte:

```
public MyClass() {}
```

- Den skapas bara om inga andra konstruktorer definieras. Så fort någon annan konstruktor definieras, oavsett signatur, så skapas ingen default constructor.
- Om en konstruktor i en sub-klass *inte* gör ett explicit anrop till någon super-konstruktor, då anropas implicit `super()`, vilket (om den inte omdefinierats) innebär default constructor.
- Felmeddelanden av typen "There is no default constructor available in ..." beror på att super-klassen definierat en konstruktor med annan signatur, medan sub-klassen fortfarande försöker implicit anropa `super()`, som inte längre skapas implicit.

Sammanfattning

Quiz

- Vad är vårt mål?
- Svar: Att ni ska lära er att skriva objekt-orienterad kod, som är
 - Lätt att underhålla (maintainable)
 - Testbar
 - Robust vid förändringar
 - Återanvändbar (reuseable)
 - Utökningsbar (extensible)
- Objekt-orienterad programmering och design

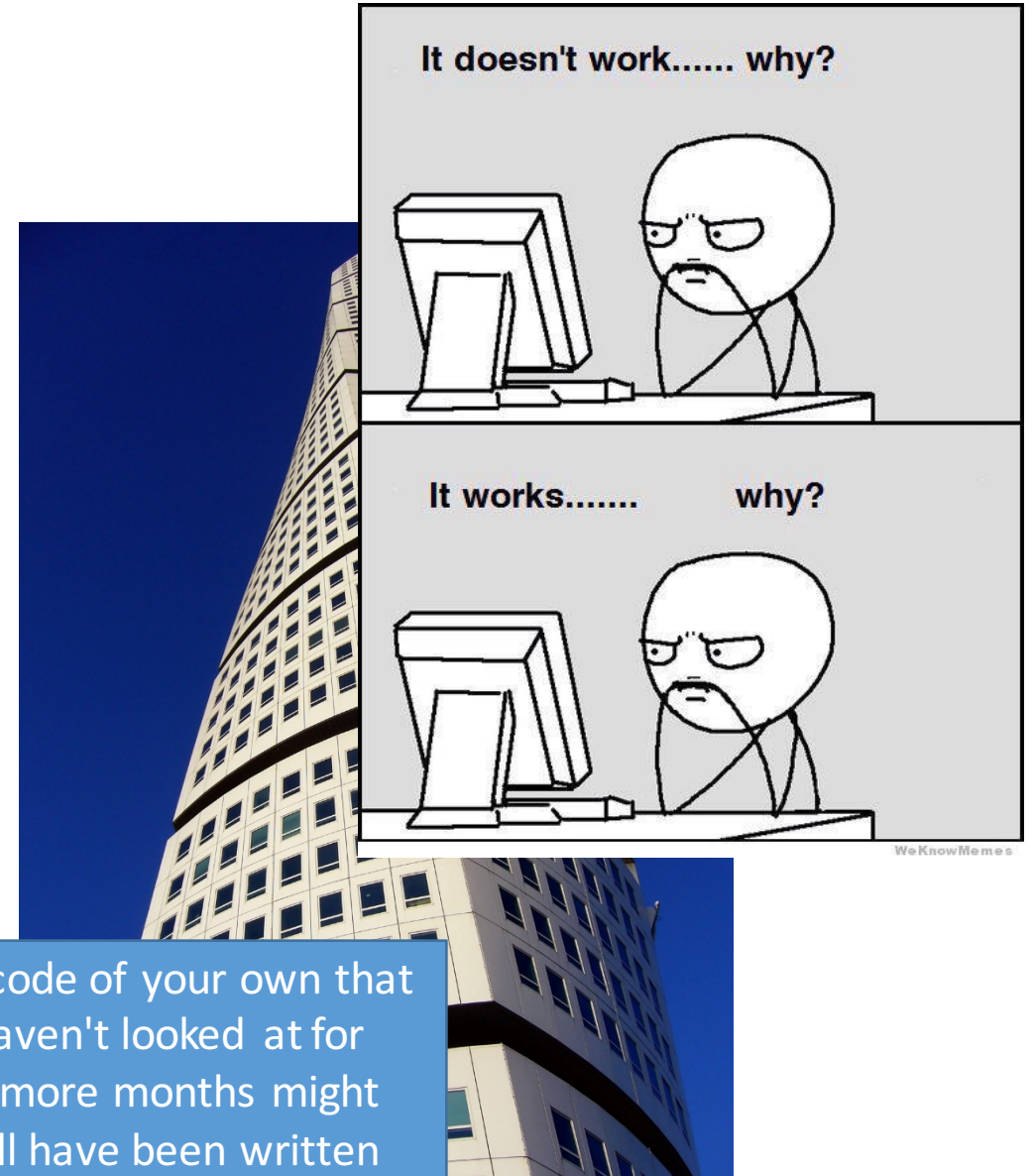
Småskalig programmering

- Triviala program
- Få klasser
- Några 100-tal rader kod
- En eller ett fåtal programmerare
- Ingen eller kort livstid
- "Just do it"



Storskalig programmering

- Mycket komplexa programsystem
- Flera miljoner rader kod
- 100-tals programmerare, geografiskt utspridda
- Lång livstid
- "Software engineering"
 - Behov av verktyg
 - Behov av processer



"Any code of your own that you haven't looked at for six or more months might as well have been written by someone else." - Eagleson's law

Objekt-orientering

- Objekt-orientering är en *metodik* för att – rätt använd! – reducera komplexitet i mjukvarusystem.
- Rätt använd ger objekt-orientering stora möjligheter till:
 - Code reuse
 - Extensibility
 - Easier maintenance
- Fel använd riskerar objekt-orientering att skapa extra komplexitet
 - "Big ball of mud"
 - Det finns mycket dålig kod därute...
 - Det finns väldigt många missförstånd kring objekt-orientering.

Objekt-orienterad modellering

- Ett program är en modell av en (verklig eller artificiell) värld.
- I en objekt-orienterad modell består denna värld av en samling objekt som *tillsammans* löser den givna uppgiften.
 - De enskilda objekten har *specifika ansvarsområden*.
 - Varje objekt definierar *sitt eget beteende*.
 - För att fullgöra sin uppgift kan ett objekt behöva *support* från andra objekt.
 - Objekten samarbetar genom att *kommunicera* med varandra via meddelanden.
 - Ett meddelande till ett objekt är en begäran att få en *uppgift utförd*.

Alla dessa kriterier ska
vara uppfyllda!

Design

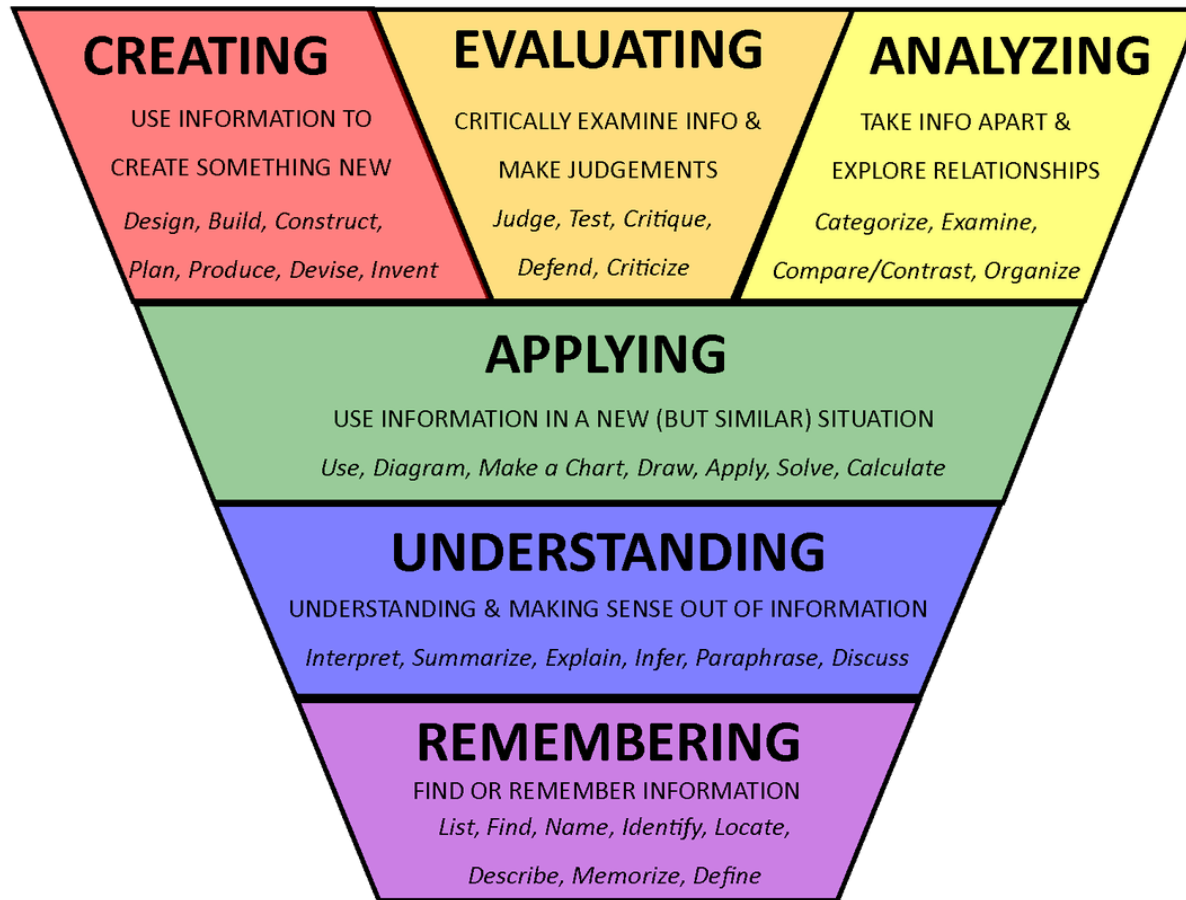
- Designen (modellen) utgör underlaget för implementationen.
 - Bra design minskar kraftigt tidsåtgången för implementationen.
 - Brister i designen överförs till implementationen och blir mycket kostsamma att åtgärda.
- Vanligaste misstaget i utvecklingsprojekt är att inte lägga tillräckligt med tid på att ta fram en bra design.
 - Bra design är svårt!!
- I allmänhet bör mer tid avsättas för design än för implementation.

Lärandemål – konkret

- Kunskap och förståelse:
 - **REDOGÖRA** för objekt-orienterade design-principer.
 - **KÄNNA IGEN** och **REDOGÖRA** för olika objekt-orienterade design-mönster, inklusiver deras syfte och effekt.
- Färdigheter och förmåga:
 - **APPLICERA** design-principer och design-mönster för att åstadkomma sund objekt-orienterad design.
 - **ANVÄNDA** och **REDOGÖRA** för grundläggande objekt-orienterade koncept, som *klasser* och *objekt*, *primitiver* och *referenser*, *metoder* och *konstruktörer*, *variabler* och *attribut*, etc.
 - **ANVÄNDA** och **REDOGÖRA** för mer avancerade språkmekanismer och tekniker, som *exceptions*, *generics*, *threads*, *defensive copying*, etc.
 - **ANVÄNDA** och **REDOGÖRA** för arv och parameteriserade typer, och därtill hörande mekanismer, för att åstadkomma polymorfism och återanvändning av kod.
 - **DESIGNA** och **IMPLEMENTERA** objekt-orienterade program för en given domän.
 - **UTFÖRA** och **BESKRIVA** testning av objekt-orienterade program.
- Värderingsförmåga och förhållningssätt:
 - **ANALYSERA** och **UTVÄRDERA** kod enligt principer för god objekt-orienterad design och implementation.

Lärandemål

Blooms lärandepyramid



DESIGNA, IMPLEMENTERA

UTVÄRDERA

ANALYSERA

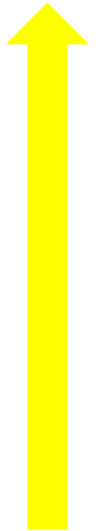
ANVÄNDA, APPLICERA

REDOGÖRA

KÄNNA IGEN



Kod



Verktyg och
Principer

Lärandemål – konkret

- Kunskap och förståelse:
 - **REDOGÖRA** för objekt-orienterade design-principer.
 - **KÄNNA IGEN** och **REDOGÖRA** för olika objekt-orienterade design-mönster, inklusiver deras syfte och effekt.
- Färdigheter och förmåga:
 - **APPLICERA** design-principer och design-mönster för att åstadkomma sund objekt-orienterad design.
 - **ANVÄNDA** och **REDOGÖRA** för grundläggande objekt-orienterade koncept, som *klasser* och *objekt*, *primitiver* och *referenser*, *metoder* och *konstruktörer*, *variabler* och *attribut*, etc.
 - **ANVÄNDA** och **REDOGÖRA** för mer avancerade språkmekanismer och tekniker, som *exceptions*, *generics*, *threads*, *defensive copying*, etc.
 - **ANVÄNDA** och **REDOGÖRA** för arv och parameteriserade typer, och därtill hörande mekanismer, för att åstadkomma polymorfism och återanvändning av kod.
 - **DESIGNA** och **IMPLEMENTERA** objekt-orienterade program för en given domän.
 - **UTFÖRA** och **BESKRIVA** testning av objekt-orienterade program.
- Värderingsförmåga och förhållningssätt:
 - **ANALYSERA** och **UTVÄRDERA** kod enligt principer för god objekt-orienterad design och implementation.

Laboration: egen kod

- Syftar till att ni ska få träna på att använda tekniska aspekter, och få en känsla för polymorfism och code reuse
 - Arv, delegering, generics(?), overriding, ...
- Testar (främst) följande lärandemål:
 - **ANVÄNDA** och **REDOGÖRA** för grundläggande objekt-orienterade koncept, som *klasser och objekt, primitiver och referenser, metoder och konstruktörer, variabler och attribut*, etc.
 - **ANVÄNDA** och **REDOGÖRA** för *arv* och *parameteriserade typer*, och därtill hörande mekanismer, för att åstadkomma polymorfism och återanvändning av kod.
 - **UTFÖRA** och **BESKRIVA** testning av objekt-orienterade program.
 - (**ANVÄNDA** och **REDOGÖRA** för mer avancerade språkmekanismer och tekniker, som *exceptions, generics, trådar, defensive copying*, etc.)

Laboration: annans kod

- Syftar till att ni ska träna på att läsa och förstå existerande kod, att anpassa egen och annans kod, och förbättra kod med hjälp av strukturerad refaktorering.
- Testar (främst) följande lärandemål:
 - **APPLICERA** design-principer och design-mönster för att åstadkomma sund objekt-orienterad design.
 - **ANVÄNDA** och **REDOGÖRA** för mer avancerade språkmekanismer och tekniker, som *exceptions, generics, defensive copying*, etc.
 - **ANALYSERA** och **UTVÄRDERA** kod enligt principer för god objekt-orienterad design och implementation.
 - **DESIGNA** och **IMPLEMENTERA** objekt-orienterade program för en given domän på ett sunt sätt med avseende på korrekthet, modifierbarhet och återanvändbarhet.

Inlämningsuppgift

- Inlämningen görs i samma grupper som tidigare.
- Testar (främst) följande lärandemål:
 - **ANALYSERA** och **UTVÄRDERA** kod enligt principer för god objekt-orienterad design och implementation.
 - **KÄNNA IGEN** och **REDOGÖRA** för olika objekt-orienterade design-mönster, inklusiver deras syfte och effekt.
 - **APPLICERA** design-principer och design-mönster för att åstadkomma sund objekt-orienterad design.
- Vid tentan kommer vi ställa kontrollfrågor kring er inlämning, för att bedöma till vilken nivå ni uppfyller dessa mål.
 - Frågorna kommer variera stort mellan olika grupper och individer.

Muntlig tentamen

- Testar följande lärandemål:
 - **REDOGÖRA** för objekt-orienterade design-principer.
 - **KÄNNA IGEN** och **REDOGÖRA** för olika objekt-orienterade design-mönster; inklusive deras syfte och effekt.
 - **REDOGÖRA** för grundläggande objekt-orienterade koncept, som *klasser* och *objekt*, *primitiver* och *referenser*, *metoder* och *konstruktörer*, *variabler* och *attribut*, etc.
 - **REDOGÖRA** för mer avancerade språkmekanismer och tekniker, som *exceptions*, *generics*, *threads*, *defensive copying*, etc.
 - **REDOGÖRA** för *arv* och *parameteriserade typer*, och därtill hörande mekanismer, för att åstadkomma polymorfism och återanvändning av kod.

Muntlig tentamen

- När: Under vecka 2, vid en tidpunkt ni själva bokar på kurshemsidan
- Var: EDIT 5471, 6471 (och 6128)
- Vem: Alex Gerdes, Niklas Broberg, Johannes Åman Pohjola

- Hjälpmedel: Inga
- Tillhandahålles: UML syntax "cheat sheet"

Roadmap

- Delmoment
 1. Tekniska verktyg och mekanismer
 2. Principer
 3. Patterns
 4. Tekniker

Tekniska verktyg

- Klasser, objekt, gränssnitt (interfaces), typer, referenser (1-2)
 - Statiska vs dynamiska typer, overriding och overloading (2-2)
 - Arv vs delegering (3-1)
 - Generics, varians(3-2, 4-1)
 - Exceptions (7-2)
 - Trådar (7-1)
-
- Polymorfism (4-1)
 - Encapsulation (4-2)

Roadmap

- Delmoment
 1. Tekniska verktyg och mekanismer
 2. Principer
 3. Patterns
 4. Tekniker

Design-principer

- **SOLID**

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

- **Generella**

- Composition over Inheritance (3-1)
- High Cohesion, Low Coupling (4-2)
- Separation of Concern (5-1)
- Command-Query Separation (5-1)
- Law of Demeter (7-1)
 - Don't talk to strangers

OPC: The Open-Closed Principle

Software modules should be open for extension, but closed for modification.
(Bertrand Meyer)

- Förändring är det enda som är konstant.
- Utveckla *framåt-kompatibelt* – förutsäg *var* förändring kommer behövas.
- Föreläsning 1-1

Dependency Inversion Principle

Depend on abstractions, not on concrete implementations.

- Genom att använda supertyper istället för subtyper kan vi *minska beroendet* av en specifik klass.
- Föreläsning 2-1, mer på 4-1

Single Responsibility Principle

*A class should have only
one reason to change.
(Robert C. Martin)*

- "A class should have *at most* one reason to change."
 - Martins formulering tar inte hänsyn till immutabilitet.
- Föreläsning 4-2

ISP: Interface Segregation Principle

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

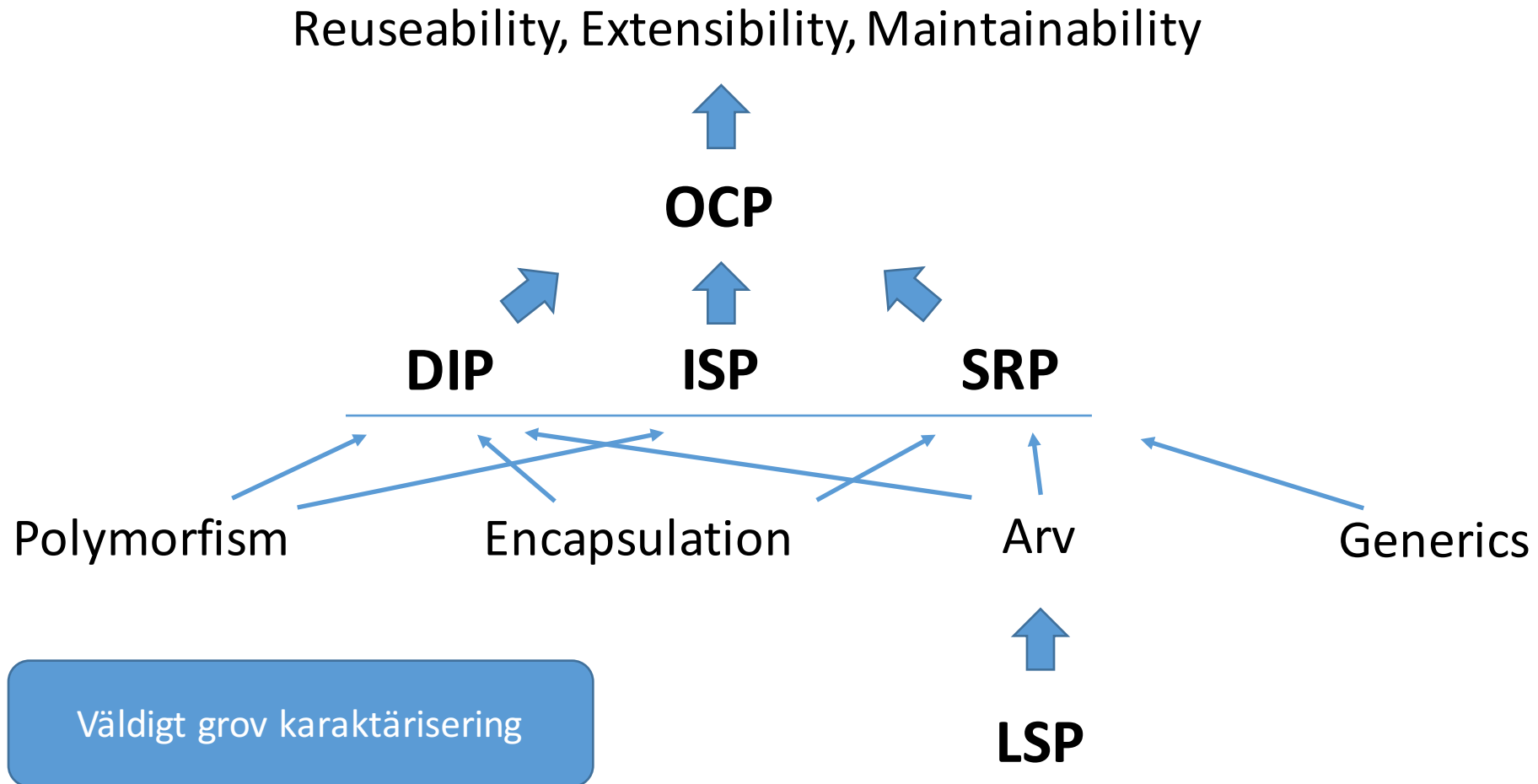
- Om ett objekt A är stort, med mycket funktionalitet, och objekt B beror på en liten del av denna funktionalitet: Introducera en abstraktion vars gränssnitt är precis den del av A som B behöver bero på.
- Föreläsning 6-2

Liskov Substitution Principle

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)

- *S* är en *äkta* subtyp till *T* endast om, för varje publik metod som finns i både *S* och *T*:
 - *S*'s metod godtar alla värden som *T*'s metod godtar
 - *S* gör alla beräkningar på denna indata som *T* gör (och kanske fler).
- Föreläsning 2-2

Principer – hur hänger de ihop?



Roadmap

- Delmoment
 1. Tekniska verktyg och mekanismer
 2. Principer
 3. Patterns
 4. Tekniker

Vad är ett design pattern?

- Ett *design pattern* (designmönster) är en (ofta namngiven) generell lösning av en vanligt återkommande situation inom (mjukvaru-)design.
 - Termen och konceptet kommer ursprungligen ifrån arkitektur.
 - Vi pratar mest om design patterns inom objekt-orienterad design, men de existerar (mer eller mindre uttalade och erkända) inom alla paradigmer.
- Ett design pattern har ingen färdig kod – de är abstrakta mallar för hur ett problem kan lösas.
 - "Formaliserade" best-practices.
 - I en given situation och kontext kan vi instansiera ett design pattern med specifika klasser, metoder, etc, och få en färdig lösning.

Creational Patterns

- Berör hur objekt skapas
- Factory
- Factory Method
- Singleton

Behavioral Patterns

- Berör hur objekt kommunicerar
 - e.g. vilka metoder och signaturer vi använder
- Observer
- Template Method, Strategy, State
- Iterator
- Chain of Responsibility

Structural Patterns

- Berör hur vi strukturerar komponenter
 - E.g. vilka klasser och interfaces vi bör använda, och hur de bör bero av varandra
- Adapter
- Bridge
- Composite
- Decorator
- Facade

Architectural Patterns

- Berör hur vi strukturerar våra program på en högre nivå än klasser
- Module
- Model-View-Controller

The Complete List (för tentan)

Adapter, Bridge, Chain of Responsibility,
Composite, Decorator, Facade, Factory,
Factory Method, Iterator, Mediator,
Model-View-Controller, Module,
Observer, Singleton, State, Strategy,
Template Method

Roadmap

- Delmoment
 1. Tekniska verktyg och mekanismer
 2. Principer
 3. Patterns
 4. Tekniker

Tekniker

- Refactoring (1-1)
- Functional Decomposition (4-2)
- Immutability (6-1)
- Defensive Copying (6-1)
- Mutate-by-copy (6-1)
- Method Cascading (6-2)

Frågor?