Föreläsning 1

Introduktion
Utveckla för förändring

Grundkursen

I grundkursen fick ni:

- lära er de grundläggande principerna för objektorienterad programmering
- lära er de grundläggande konstruktionerna i programspråket Java
- bekanta er med klassbiblioteket i Java
- konstruera enkla program.

Koncept som är kända från grundkursen (?)

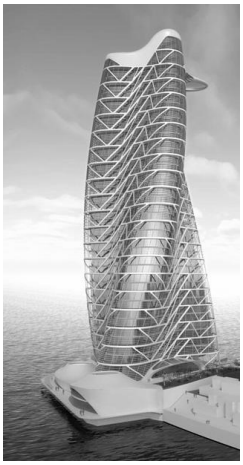
klasser	objekt	typer
typomvandling	enkla variabler	omslagsklasser
uppräkningstyper	räckvidd	referensvariabler
klassvariabler	instansvariabler	attribut
synlighetsmodifierare	inkapsling	konstruktorer
metoder	instansmetoder	klassmetoder
överlagring	värdeanrop	referensanrop
metodsignatur	arv	superklass
subklass	association	relationer
överskuggning	polymorfism	dynamisk bindning
klasshierarki	abstrakta klasser	abstrakta metoder
interface	anonyma klasser	paket
UML	strömmar	exceptionella händelser
m.m		

Programming in the small

- triviala program
- några 100-tal rader kod
- en eller ett fåtal programmerare
- ingen eller kort livstid
- ”programmera direkt”-metoden

Programming in the large

- mycket komplexa programsystem
- kanske flera miljoner rader kod
- kanske 100-tals programmerare som är geografiskt utspridda
- lång livstid
- software engineering
  - behov av utvecklingsverktyg
    kodstandard
    testverktyg
    versionshantering
    . . .
  - behov av utvecklingsprocesser
  - . . .



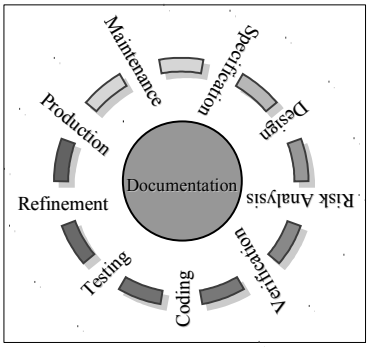
Mjukvarans livscykel

Kommersiella programsystem har lång livstid och kraven på systemen förändras under deras livstid. Ny funktionalitet läggs till.

Ingen person kan vara insatt i och förstå alla enskilda delar i systemet.

Andra programmerare än de som utvecklade systemet utför uppdateringar och systemunderhåll.

En stabil design och en bra dokumentation är mycket viktigt.



Utmaningen med utveckling av mjukvarusystem

När man utvecklar ett mjukvarusystem finns en rad krav och önskemål som skall uppfyllas. Vi vill åstadkomma ett system som:

- är färdigutvecklat på utsatt tid och inom givna budgetramar
- uppfyller sin uppgift så bra som möjligt
- är så enkelt att använda som möjligt
- har tillräckligt höga prestanda
- är felfritt
- är robust
- är så enkelt att underhålla som möjligt
- är så enkelt att modifiera som möjligt
- är så enkelt att bygga ut som möjligt
- är så enkelt att implementera som möjligt
- är så återanvändbart som möjligt
- . . .

Mjukvarukvalité

Attribut som är viktiga och synliga för användarna (externa faktorer):

• Användbarhet (usability)	är systemet lätt att använda?
• Korrekthet (correctness)	fungera systemet i enlighet specifikationen?
• Robusthet (robustness)	klarar systemet av oförutsedda situationer utan att krascha?
• Tillförlitlighet (reliability)	systemet är tillförlitligt om det är korrekt och robust.
• Fullständighet (completeness)	uppfyll systemet användarnas behov?
• Tillgänglighet (availability)	hur ofta går systemet ner?
• Effektivitet (efficiency)	klarar systemet av att ge efterfrågad service inom rimlig tid och med rimliga resurser?
• Flexibilitet (flexibility)	kan systemet anpassas för individuella behov?
• Skalbarhet (scalability)	kommer systemet att fungera korrekt även om om det problem som handhas skalas upp en magnitud?

Mjukvarukvalité

Attribut som är viktiga och synliga för utvecklarna (interna faktorer):

• Underhållsbarthet (maintainability)	är systemet lätt att modifiera och utöka med ny funktionalitet?
- Läsbarhet (readability)	är koden lätt att läsa och förstå?
- Enkelhet (simpilicty)	är koden onödigt komplex?
- Utbyggbarhet (extensibility)	är det enkelt att lägga till nya tjänster och ta bort gamla?
• Återanvändbarhet (reusability)	kan komponenter i systemet lätt återanvändas inom systemet eller i andra system?
• Testbarhet (testability)	är det lätt att testa systemet och de ingående komponenterna?

Det är främst utvecklarnas aspekter på mjukvarukvalité som vi kommer att behandla under kursen.

Underhåll av mjukvarusystem

> *"All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version." (Ivar Jacobson)*

Att ett programsystem skall vara lätt att underhålla är en mycket viktig egenskap!

- Fel i den ursprungliga mjukvaran måste fixas.
- Mjukvaran måste anpassas till nya användarkrav.
- Den som ändrar i koden är sällan eller aldrig den som ursprungligen skrivit koden.
- Största delen av pengarna och tiden under ett systems livstid åtgår till systemunderhåll (ca 80%).

Utveckla för förändring!

Utveckla för förändring

Vi lever i en föränderlig värld och vet med säkerhet att förändringar kommer att ske i de programsystem vi utvecklar.

Däremot vet vi inte med säkerhet *vilka* förändringar som kommer att bli aktuella.

Vi kan dock ofta någorlunda bra förutsäga *var* ändringarna kommer att införas och förbereda för detta – d.v.s. göra vårt system *framåtkompatibelt*.

Vi behöver identifiera de delar av systemet som kommer att påverkas av framtida ändringar eller utökningar och förbereda genom att så långt som möjligt isolera och frigöra dessa delar från det övriga systemet.

> ***The Open-Closed Principle (OPC):***
>
> *Software modules should be open for extension, but closed for modification. (Bertrand Meyer)*

Det objektorienterade paradigmet

Objektorientering är en metodik som utvecklats specifikt för att reducera komplexiteten i mjukvarusystem.

Rätt använd har objektorientering stor potential till:

- återanvänding av kod
- utbyggbarhet
- enklare systemunderhåll

Fel använd riskeras att få *"a big ball of mud"*, i vilken man sitter fast i utan att kunna göra något produktivt.

Design smells: Symtom på dålig design

Stelhet (*rigidity*) – när en ändring är svår att göra eftersom varje ändring propagerar till många andra delar av systemet och ger upphov till en mängd av ändringar i beroende moduler.

Bräcklighet (*fragility*) – när en ändring gör att det uppstår nya fel i delar av systemet som inte har något konceptuell koppling till den ändrade modulen.

Orörlighet (*immobility*) – koden är svår att återanvända i andra applikationer. Modulerna är så hårt kopplade till varandra att de inte kan frigöras från den aktuella applikationen.

Seghet (*viscosity*) – man har gjort designval som innebär att det krävs stora insatser att göra en "bra" ändring, varför lösningen blir ett "hack".

Oklarhet (*opacity*) – en modul är svår att läsa och förstå.

Orsaker till design smells

"A system that is used will be changed. An evolving system increases its complexity unless work is done to reduce it." (Manny Lehman)

Förruttnelsen orsakas direkt eller indirekt av ***olämpliga beroenden*** mellan de ingående delarna i systemet:

- Den ursprungliga designen är undermålig:
  - bristfällig förståelse av de ursprungliga kraven
  - bristfällig förståelse av systemets framtida utveckling
  - inkompetenta utvecklare
- Ändrade krav beaktas inte på allvar
  - snabba hack görs under tidspress för att lösa akuta problem
  - de som gör ändringarna förstår inte den ursprungliga designen, varför ändringar blir svåra att göra och de ändringar som görs ökar komplexiteten hos designen
  - nya beroenden mellan komponenter skapats.

Design smells

Hur motverka design smells

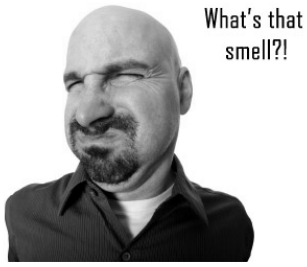
- Inse från början att kraven kommer att förändras
  - kräver ändringar i systemet.
- Använd beprövade tekniker för att minska beroenden mellan komponenterna i systemet.
- Refaktorisering (refactoring)
  - omstrukturering av koden som bevarar funktionalitet men förbättrar strukturen, utan att påverka det yttre beteendet
  - kvalitén på designen skall alltid hållas hög under systemets hela livstid
  - det är omöjligt att designa ett perfekt system på första försöket.

Unken kod (Code smells)

Synliga tecken i koden på att designen håller på att ruttna:

- Duplicerad kod, “klipp&klistra”-programmering
- Långa metoder
- Långa parameterlistor
- Stora klasser
- Klasser som endast innehåller data
- Publika instansvariabler
- Långa **switch**-satser
- Avsaknad av kommentarer
- Onödiga kommentarer
- Oläslig kod

…



Complexity is your enemy

> *A person can only keep 7 plus or minus 2 items in mind at one time. (George Miller)*

Mycket i programmering handlar om att reducera komplexiteten hos den mjukvara som utvecklas.

> *Design and programming are human activities; forget that and all is lost. (Bjarne Stroustrup)*

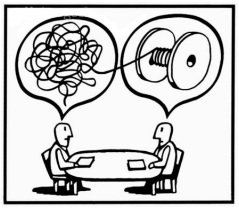
Komplexitet i ett program handlar inte om hur bra ett program går att köra på datorn, utan helt och hållet om hur lätt det är att läsa och förstås av människor.

Utveckla för förändring

> *Controlling complexity is the essence of computer programming. (Brian Kernighan)*

För att åstadkomma programvara som blir enkel att underhålla och återanvända måste man försöka reducera komplexiteten på alla nivåer i utvecklingsarbetet, allt ifrån design av den övergripande systemarkitekturen till val av variabelnamn.

Man måste hela tiden vara medveten om problemet med komplexitet och lära sig *best practices* för hur komplexitet skall hanteras.



Grundläggande designprinciper

- Enhetlig kodstil. Minskar risken för fel och underlättar förståelsen av koden.
- Genomtänkt namngivning. Ett namn skall avspegla vilken roll en entitet har.
- Dokumentation. Oavsett hur bra design en mjukvara har är den värdelös utan dokumentation.
- Modularitet. Ett system skall brytas ned till en samling sammanhängande men löst kopplade moduler.
- Abstraktion. Bortse från detaljer när de är irrelevanta.
- Inkapsling & döljande av information. *Keep it secret, keep it safe.*
- Decoupling. Minimera beroenden mellan klasser, minimera beroenden av specifika typer.
- Delegering. Skicka vidare uppgifter till de klasser som är bäst lämpade att göra arbetet.
- Designmönster. Använd välkända och accepterade lösningar.
- Återanvändning av kod. *Don't Repeat Yourself* (DRY).

Kodstil och namngivning

> *Programs should be written for people to read, and only incidentally for machines to execute. (Abelson och Sussman)*

Konstruera och underhålla programvara är en process som sker mellan människor!

För att underlätta för andra att läsa den kod du utvecklar skall du använda en kodkonvention. Lämpligen

> http://java.sun.com/docs/codeconv

En av de viktigaste aspekterna på kodkonvention berör namngivning.

Identifierarnamn måste väljas mycket omsorgsfullt.

Namnet är det första läsaren ser och skall vara en ledtråd för att förstå vad identifieraren representerar och vilken uppgift den har.

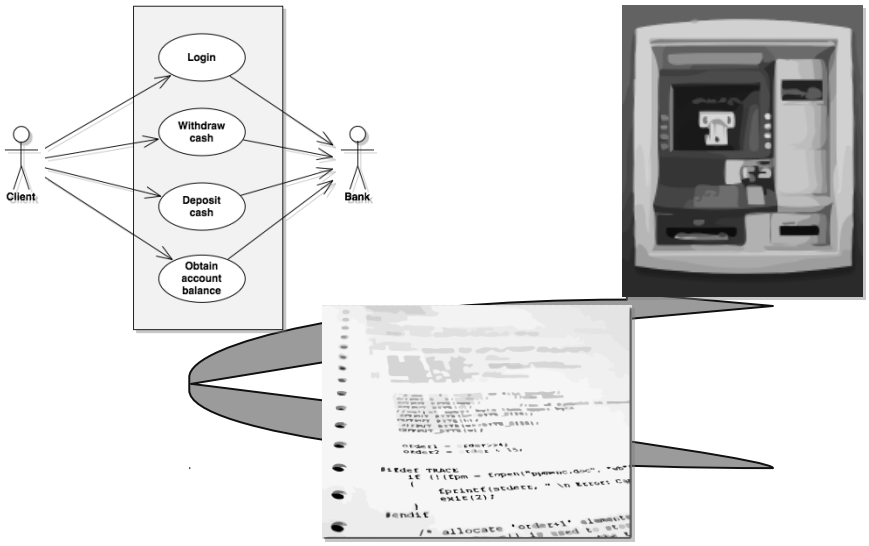
Let the code talk!

Kodstil och namngivning

Klasser:	Namnen är oftast substantiv, t.ex Date, BankAccount och Car.
Interface:	Namnen slutar ofta på ”able” , t.ex Cloneable och Comparable.
Metoder:	Namnen skall vara ett verb eller innehålla ett verb, t.ex. print, setSize, isVisible och getSize.
Variabler:	numberOfMembers och loadingDone är bra namn. theNumberOfThreadsInTheApplicationRunningRightNow, nT, notYetDone och done är dåliga namn.

Använd engelsk namngivning.

Programmering är modellering



Programmering = modellering

Ett program är en modell av en verklig eller tänkt värld.

- I objektorienterad programmering består denna värld av ett antal objekt som tillsammans löser den givna uppgiften.
  - De enskilda objekten har specifika ansvarsområden.
  - Objekten samarbetar genom att kommunicera med varandra via meddelanden.
  - Ett meddelande till ett objekt är en begäran från ett annat objekt att få något utfört.

Att göra en bra modell av verkligheten, och därmed möjliggöra en bra design av programmet, är en utmaning.

Modulär design

*Målsättningen skall vara att designa programsystemet runt **stabila abstraktioner** och **utbytbara komponenter** för att möjliggöra små och stegvisa förändringar*.

Modulär design

Ett programsystem är för stort för att kunna förstås i sin helhet

- måste dela upp systemet i mindre delar. Denna process kallas för *dekomposition*.

Ett modulärt system är uppdelat i identifierbara abstraktioner som kallas moduler (t.ex. klasser, paket och subsystem).

Fördelar med en *välgjord* modulär design:

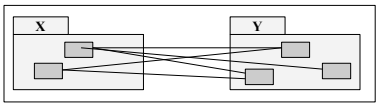
- lätt att utvidga
- moduler går att återanvända
- uppdelning av ansvar
- komplexiteten reduceras
- moduler går att byta ut
- tillåter parallell utveckling.

Koppling (coupling)

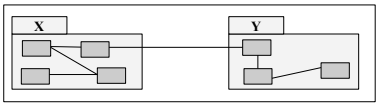
För att få en flexibel design måste de ingående modulerna vara så oberoende av varandra som möjligt.

Med koppling avses *hur starkt* beroendet är mellan två olika moduler.

Stark koppling



Svag koppling



Har vi starka kopplingar kommer förändringar i en modul att framtvinga förändringar även i de moduler som är beroende av denna.

Desto lägre grad av koppling som finns mellan modulerna i ett system, desto mer flexibelt och förändringsbart blir systemet.

Koppling (coupling)

För att de ingående modulerna i ett system skall kunna samarbeta vid lösandet av en uppgift måste det naturligtvis finnas kopplingar mellan modulerna.

Men för att få en bra design är vår uppgift att:

- minimera antalet kopplingar
- ha så lösa kopplingar som möjligt
- ha kontroll över kopplingarna
  - varje kopplingspunkt skall vara avsiktlig
  - varje kopplingspunkt skall ha ett väldefinierat gränssnitt.

Dependency Inversion Principle (DIP):

Depend on abstractions, not on concrete implementations.

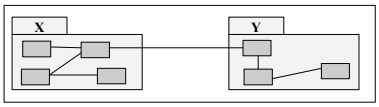
Kohesion (cohesion)

Kohesion är ett mått på den inre sammanhållningen i en modul, d.v.s. det inbördes beroendet mellan komponenterna i modulen.

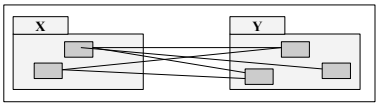
En modul har hög kohesion när komponenterna i modulen sinsemellan samverkar med varandra för att lösa den uppgift modulen har, utan att behöva samverka med komponenter i andra moduler.

Kohesion är dels ett mått på hur autonom en modul är, dels ett mått på hur väl definierad modulens ansvarsområde är.

Hög kohesion



Låg kohesion



Återanvändning

Låg koppling och hög kohesion lägger grunden för återanvändning.

Ju mer välspecificerad uppgift ett subsystem har och ju enklare dess gränssnitt är, desto större chans är det att subsystemet kommer att återanvändas.