TDP004 Objektorienterad Programmering
Fö 7 Objektorienterad design, tips och råd

Introduktion

- Vi har diskuterat vilka möjligheter till OO som erbjuds i C++.
- Vilka vill vi använda och varför? Allt har användningsområden men är de samma som problemet du vill lösa?
- ”Om man bara har en hammare så verkar alla problem vara en spik”.
- Efterföljande speglar mina erfarenheter och frågor till flera verksamma C++programmerare.
- Inte tentamensmaterial, men användbart ändå.

”Lagom är bäst”

- Ny teknik favoriseras pga nyhetens behag. En bra design använder den metod/design som passar bäst för ändamålet.
- Det är inte säkert att OO är det bästa i din situation…
- I framtiden kanske vi går tillbaka till att använda mera procedurell programmering pga trådar (multi core CPU:er, assymetriska processorer etc) med olika exekveringshastigheter.

OO eller ej?

- När ska man använda OO?
  - Omöjligt att svara på.
- Vad tillåter språket?
- Finns det något som kan vara ett objekt?
  - Har objekten egenskaper och data?
  - Logiskt att spara dem i samma objekt?
- Potentiella subklasser?
- Ovanstående är bara godtyckliga bedömningar.

OO(?) exempel
En beskrivning av ett system

- ”I en bank finns det olika sorters konton; sparkonto, långtidskonto och korttidskonto. Ett konto har minst en innehavare. Varje innehavare kan ha flera konton. På alla sorters konton kan man sätta in och ta ut pengar samt kontrollera innehavet. Kontona har olika räntor och denna ska beräknas på olika sätt enligt kap 2.4. Transaktionshistoriken ska sparas, och denna ska ha det innehåll som beskrivs i kap 3.1.”

En beskrivning av ett system, forts.

- Kap 2.4.
  - Räntor för olika sorters konton ska beräknas månadsvis på sparkonto, årsvis på långtidskonto och dagsvis på korttidskonto. Räntorna är olika för olika sorters konton och ändras genom den existerande räntehanteraren.
- Kap 3.1
  - En transaktion ska innehålla summan som blev flyttad/insatt/uttagen samt vilket eller vilka konton som är involverade.

Från beskrivning till klassdiagram

- I sin enklaste form kan ett objekt i en beskrivning bli en klass, en aktivitet blir en funktion och mängdförhållanden representerar just det.
- Objekt
- Aktiviteter
- Mängdförhållanden

Identifiera klasser mm

- ”I en bank finns det olika sorters konton: sparkonto, långtidskonto och korttidskonto. Ett konto har minst en innehavare. Varje innehavare kan ha flera konton. På alla sorters konton kan man sätta in och ta ut pengar samt kontrollera innehavet. Kontona har olika räntor och denna ska beräknas på olika sätt för olika konton enligt kap 2.4. Transaktionshistoriken ska sparas, och denna ska ha det innehåll som beskrivs i kap 3.1.”

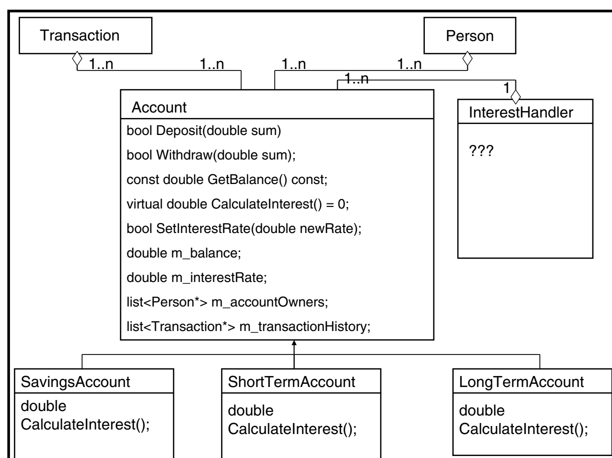
Identifiera klasser mm, forts.

- Kap 2.4.
  - Räntor för olika sorters konton ska beräknas för varje månad på sparkonto, för varje år på långtidskonto och för varje dag på korttidskonto. Räntorna är olika för alla sorters konton och ändras genom den existerande räntehanteraren.
- Kap 3.1
  - En transaktion ska innehålla summan som blev flyttad/insatt/uttagen och vilket eller vilka konton som är involverade.

Klasser och aktiviteter

- Konton: sparkonto, långtidskonto, korttidskonto
- Innehavare
- Transaktionshistorik
- Räntor
- Räntehanteraren
- Sätta in/ta ut pengar, kontrollera innehavet
- Ändra ränta
- Beräkna ränta
- Spara transaktionshistorik

- Partiellt klassdiagram i fusk-UML på nästa bild.

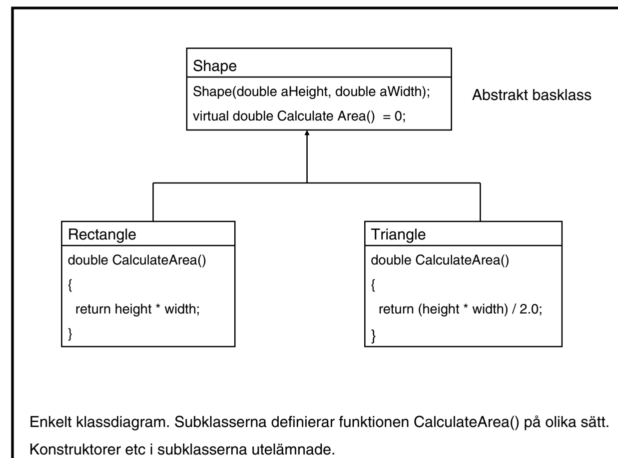


OO exempel, diskussion

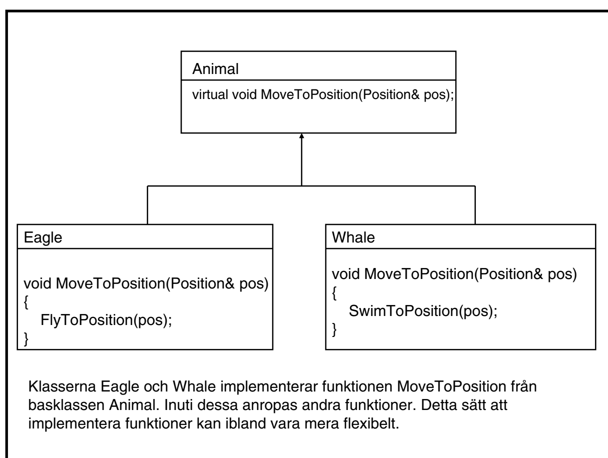
- Räntan – ingen egen klass.
  - Bara en variabel. Inga egna funktioner.
- InterestHandler en svart låda.
- Transaktionshistoriken – list<Transaction*>. Kunde ha varit TransactionHistory* med annan TransactionHistory-klass.
- Naturligt med arv från Account?
- Naturligt att Person är en fristående klass?
  - Saknas funktioner för att ta bort/lägga till Person.
- Lätt exempel – ingen entydig uppdelning.

Arv

- Tänk på virtuell destruktor i basklassen.
- Abstrakta klasser har användningsområde.
- Virtuella funktioner används med fördel då subklasserna utför beräkningarna på olika sätt.
- Polymorfi är kraftfullt. Även fördelaktigt om man är flera som parallellt programmerar på olika subklasser till samma superklass.
- Skilj på virtuella och rent virtuella funktioner.



Enkelt klassdiagram. Subklasserna definierar funktionen CalculateArea() på olika sätt. Konstruktorer etc i subklasserna utelämnade.



Klasserna Eagle och Whale implementerar funktionen MoveToPosition från basklassen Animal. Inuti dessa anropas andra funktioner. Detta sätt att implementera funktioner kan ibland vara mera flexibelt.

Abstrakta klasser

- Har minst en rent virtuell funktion
- Omöjligt att skapa en instans av en abstrakt klass.
- När ska man använda abstrakta klasser?
  - När det verkar ologiskt att kunna skapa en instans av något (tex djur på föregående bild. Vad representerar ett generellt djur?)

Åtkomst

- Fundera noga över vad som ska vara public och private.
- Vad av det som är private kan tänkas vilja ärvas från den här klassen? Dessa är *kandidater* till att vara protected.
- Friend används med försiktighet. Bra ibland men långt ifrån alltid.

Inkapsling

- **Använd möjligheterna till inkapsling.**
- Göm implementationen i en funktion, returnera const & / const* / const <typ> om returvärdet inte ska kunna ändras.
- Det som du vill att andra ska kunna titta på/ändra är public. *Public data kan ändras utan att den ägande klassen vet om det.*
- *Private data och public gränssnitt (t.ex. get/set).*
- Data/funktioner som ingen annan ska använda ska vara private. Tex hjälpfunktioner till den egna klassen.

Inkapsling, forts.

- **Fördelar**
  - Viktiga interna data är mera skyddade från misstag från andra klasser.
  - Implementationen kan utvecklas utan att interfacet ändras. Dvs du kan ändra hur du löser en uppgift utan att någon utomstående behöver veta om det. Viktigt i större projekt.
  - Minskar beroendet mellan klasserna: ingen utomstående vet vilka variabler som finns och kan inte vara beroende av dem.

Composition vs Inheritance (”Has A” versus ”Is A”)

- Ska en klass vara en subklass till en annan eller ska den tillgång till information från andra klasser?
- Läs beskrivningen av klasserna:
  - Hålls de logiskt ihop?
  - Verkar de beskriva liknande saker?
  - Verkar det en finnas en vettig arvshierarki?
- *Ofta föredrar man composition istället för inheritance.*
- Arv orsakar ibland problem, dessa undviks ofta mha composition. Tänk på inkapsling och programmera defensivt.

Separera uppgifter

- Lätt hänt att en klass blir en hel verktygslåda som försöker lösa alla problem.
- Separera (arv och/eller composition) till flera klasser som var och en löser färre uppgifter vardera.
- Undersökningar visar att en klass komplexitet (och därmed buggarna) ökar *kvadratiskt* med antalet uppgifter i klassen.
- Gör en sak och gör den bra!

Modularitet

- En klass/enhet/subsystem gör sin sak och har tillgång till den data och de funktioner som det behöver för att lösa det.
- Om möjligt – försök att ”knyta ihop” några klasser som sköter samma uppgifter/hanterar samma data till ett mindre system. Detta mindre system har sedan väldefinierade interface till resten av programmet.

Cohesion

- Cohesion mäter hur ”fokuserade” och sammanhängande klasserna i en modul är.
- Exempel på låg cohesion:
  - Modul C använder endast data från modul D.
  - Klasserna i modul C har ingen med varandra och göra.
- Exempel på hög cohesion:
  - Klasserna arbetar tillsammans för att utföra en gemensam uppgift.
- Sträva efter hög cohesion.

Coupling

- Coupling mäter hur klasser är beroende av/påverkar varandra.
- Exempel på hög coupling:
  - Klass A påverkar B.
  - Svårt att förstå klass A ensam.
  - Svårt att återanvända klass A eftersom den kräver klass B.
- Sträva efter låg coupling.

Designmönster

- Design Patterns – standardiserad arkitektur-lösning med standardiserad terminologi, på ofta förekommande arkitekturproblem.
- Jämför med tex sorteringsalgoritm - standardiserad algoritmlösning på ofta förekommande algoritmproblem.
- Många olika patterns: Singleton, Adapter, Wrapper etc.
- Boken ”Design Patterns” av Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Utgiven av Addison Wesley.

Create-Init

- Konstruktorn returnerar inget. Gick viktiga initieringar bra? Problemet kan lösas mha Create-Init.

```
                                  // I MyClass.h
static MyClass* Create(…);
MyClass();                        // Konstruktor
bool Init(…)                      // I Init(…) sker viktiga initieringar!
                                  // I MyClass.cc
Myclass* MyClass::Create(…)
{
   MyClass* res = NULL;
   res = new MyClass();           // Konstruktorn tar ingen this som parameter.
   if (false == res->Init(…))
    {
      delete res;
      res = NULL;
   }
   return res;
}
```

Typkonverteringar

- Gör typkonverteringar om det behövs.
- Bättre att konvertera en gång för mycket än en gång för lite.
- Konvertera åt rätt håll, oftast till decimaltal/högre noggrannhet.
- Avsaknad av konverteringar kan ge svårfunna buggar och kod som ser rätt ut.

Tips från verkligheten

- STL är väldigt kraftfullt.
- const. Allt som kan bör vara const: variabler, returvärden, inparametrar och funktioner. Kompilatorn är duktig på att hitta fel.
- Vettiga variabel-, funktions- och klassnamn.
  Ett (skräck)exempel ur det verkliga livet:
  float BerBrf(int hix, int mix, int pix) ?!?!
- Vettiga och korrekta kommentarer.

Tips från verkligheten, forts.

- "The first 90% of the code accounts for the first 90% of the development time. The remaining 10% of the code accounts for the other 90% of the development time."
- Hofstadter's law: "It always takes longer than you expect, even when you take into account Hofstadter's law."
- Stens lag: ”90% av alla programmerare är 90% klara 90% av tiden.”

Sammanfattning

- Composition är ofta att föredra framför arv men även arv har sin plats.
- Modularitet och inkapsling.
- Använd const och STL.
- Designvalen är sällan svartvita.
- Det tar tid att bli bra på programmering och design – öva!