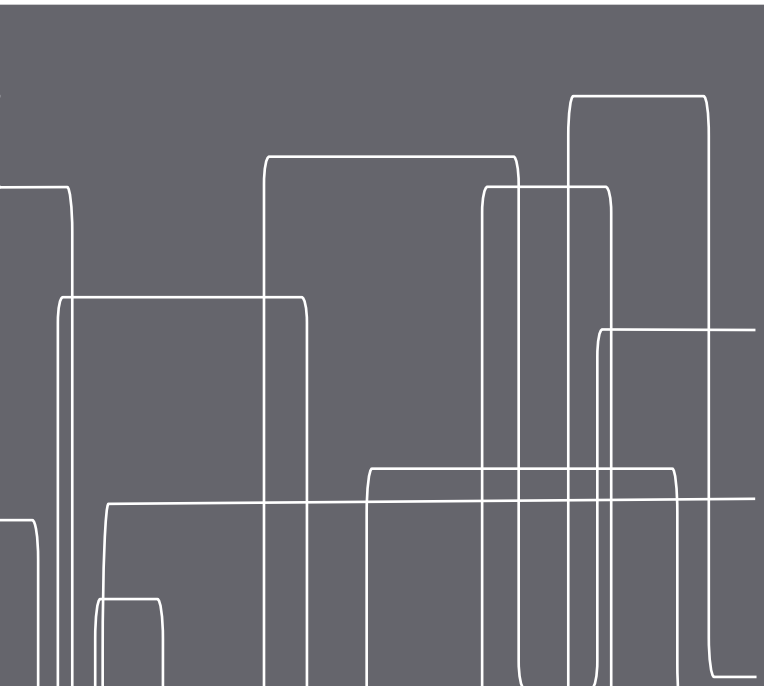
Objektorienterad Programkonstruktion

Föreläsning 15

30 jan 2017



Felsökning

- Med moderna programmeringsverktyg är rena syntaxfel oftast lätta att åtgärda
- Fel som kan vara svårare att åtgärda är t.ex:
  - thread intereferance och deadlock
  - en metod "räknar fel"
  - koden ger oönskade resultat för vissa (ovanliga) indata
  - ett metodanrop medför oönskade bieffekter
  - programmet förbrukar onödigt mycket resurser

Felsökning – Steg för steg

- Identifera att något är fel
- Återskapa felet
- Isolera felet
- Laga felet
- Verifiera att koden fungerar

Verktyg för felsökning

- *printf debugging* - man skriver ut värdet på variabler och/eller delresultat när programmet körs
- debug-verktyg, kan köra programmet fram till en given punkt, en s.k. break point. När denna punkt nås stannar programmet, och man kan läsa av värdet på de variabler man är intresserad av, och sedan antingen bryta programmet, stega vidare rad för rad, eller köra det fram till nästa break point.
- I Netbeans kan man sätta break points för specifika rader, men även för metodanrop, variabelåtkomster, klasser, mm.
- JUnit kan användas för att skapa tester som kontrollerar att koden beter sig som man väntar

Refactoring

- Med hjälp av refactoring kan man hitta felkällor
- Till exempel kan man byta direkt fältåtkomst mot get/setmetodpar, och lägga in kod som varnar eller bryter för vissa värden
- Genom att byta ut en metod mot en annan med samma funktion kan man se om fel fanns internt i metoden
- NetBeans 7 och uppåt har stöd för automatiserad refactoring

Heisenbugs

- Uppkallade efter kvantmekaniska principer att observation påverkar det objekt man observerar
- Vissa typer av fel uppstår bara när man observerar dem, alternativt endast när man inte observerar dem.
- Typiskt för parallella program, där hastigheten på en tråds exekvering påverkas av om man skriver ut variabelvärden. Beroende på två trådars relativa exekveringshastighet kan de nå en viss programrad i olika ordning.

Ovanliga fel

- Vissa program ger väntade resultat för de flesta indata, men oväntade resultat för ett begränsat antal specialfall
- Uppstår ofta för värden som ligger i extremer av tillåtna intervall
- Ibland kan extremerna uppstå lokalt inuti en beräkning, som t.ex i ett trigonometriskt uttryck där vissa funktioner är ickekontinuerliga (t.ex tan)
- Små värden kan avrundas till noll
- Program med slumptal kan träffa på oväntade resultat med låg sannolikhet som är svåra att återskapa
- Indata från användare eller mätutrustning kan betraktas som slumptal i detta avseende

Profilering

- För att göra koden effektivare måste man först veta vilken del av koden som förbrukar mest resurser
- Det är sällan värt besväret att optimera kod som står för en väldigt liten del av programmets resursförbrukning
- Verktyg som sammanställer information om ett programs resursförbrukning kallas för profilerare

Instrumenting Profiler

- Man lägger till explicita extra anrop i koden för att utföra de nödvändiga mätningarna.
- Exempel:
  - Räkna hur många gånger en metod anropas genom att räkna upp ett index
  - Ta tid på hur lång tid en metod tar att utföra, genom att jämföra systemklockan före och efter ett anrop
- Fördelar:
  - Går att utföra för alla program i alla miljöer
  - Man kan explicit mäta de saker man är intresserad av
- Nackdelar
  - Ändringarna i koden kan påverka resursförbrukningen
  - Komplexa program kräver mycket extra kod
  - Man kan missa att mäta relevanta delar

Statistical (Sampling) Profiler

- Ett externt program använder systemanrop för att göra regelbundna provtagningar av programpekaren
- Efter en körning sammanställs statistik för hur ofta pekaren befinner sig i olika metoder, för att få en bild av hur stor andel av tiden som tillbringas i olika metoder
- Fördelar:
  - Liten inverkan på programmets exekvering
  - Mäter samtliga metoder i programmet
  - Kräver liten insats av programmeraren
- Nackdelar:
  - Ger bara ett statistiskt mått, med felmarginal som ökar med provtagningsperioden
  - Dålig tillförlitlighet för metoder som tar liten del av totaltiden, eller som använder systemanrop och eller atomiska anrop
- **gprof** är ett känt exempel

Event-based Profiler

- Programmet skickar events varje gång en metod anropas, och miljön kan hantera dessa för att explicit mäta saker som tid- och minnesåtgång
- I Java finns stöd för event-baserad profilering, t.ex som plug-in till NetBeans, där JVM hanterar events
- Fördelar
  - Deterministisk mätning, ingen felmarginal
  - Alla anrop mäts, även de som skulle kunna sätta externa mätningar ur spel (atomiska anrop, systemanrop)
- Nackdelar
  - Kan påverka programmets exekvering, så att det beter sig annorlunda när man mäter det

Tester

- Kontrollera att koden ger de resultat man förväntar sig
- T.ex
  - ```
    if(MyMath.cosine(PI) != -1)
    throw new BadCosineException();
    ```
- Undersök de fall där man vet vad man förväntar sig
- Undersök de fall där resultatet spelar roll
- Undersök de fall där man förväntar sig problem
- Skriv testerna först, och därefter koden man vill testa
    ```