Om problemanalyse

Peter Sestoft

2009-02-03

1 Formål

Denne note forklarer og illustrerer begrebet *problemanalyse*. En problemanalyse præsenterer mulige løsninger på et givet problem, og diskuterer fordele og ulemper ved hver mulig løsning.

Problemanalysekapitlet i en rapport over et softwareudviklingsprojekt behandler typisk en lang række delproblemer, fx arkitektur af det samlede softwaresystem, valg af algoritmer, valg af databaseserver, design af databasetabeller, opbygning af brugergrænseflade, osv.

Det er meget normalt at studerende (og andre kommende softwareudviklere) har svært ved at skrive en problemanalyse: Når man er kommet i tanke om én løsning til et problem, hvorfor så finde på flere? Ville det ikke være bedre at programmere den løsning med det samme? Eller man kan simpelthen have svært ved at se at problemet kan løses på nogen anden måde når man er begejstret for si første idé.

Men hvis man sætter sig ned og beskriver både det problem man løser og den løsning man har fundet, vil man næsten altid finde nogle svagheder ved løsningen, og nogle alternative løsninger. De alternative løsninger er måske bedre i én henseende (fx mere effektive) men dårligere i en anden henseende (fx vanskeligere at programmere korrekt, eller vanskeligere at teste). Hvis man på skrift beskriver fordele og ulemper ved de forskellige løsninger har man et glimrende udgangspunkt for at tage en bevidst og velovervejet beslutning, i stedet for ubevidst gradvis at male sig op i et hjørne hvor man til sidst i projektet må konkludere at det er bedst at begynde helt forfra.

Resten af denne note giver to eksempler på problemstillinger og en tilhørende problemanalyse.

Det første eksempel (afsnit 2) er kunstigt og behandles helt overdrevent grundigt, men det illustrerer at selv et "indlysende" problem har løsninger af vidt forskellig natur og kvalitet.

Det andet eksempel (afsnit 3) stammer fra et faktisk projekt om et ret komplekst stykke software. Diskussionen er meget indforstået (der mangler ca. 40 siders forudgående forklaring), men meningen er heller ikke at man skal følge de tekniske detaljer, men hæfte sig ved stilen: hver mulig løsning skitseres kort, men grundigt nok til at forfatteren har belyst relevante aspekter, og derefter kommer der en kort vurdering; til slut en sammenfatning af den valgte løsning.

2 Eksempel 1 på problemanalyse: Løsning af andengradsligning

Dette eksempel er helt overdrevet detaljeret, men tjener til at vise at selv "åbenlyse" opgaver har godt af at blive gennemtænkt. Problemanalyse handler for det meste om mere overordnede aspekter af et program

2.1 Problemstilling og baggrund

Der skal programmeres en metode `solveQuadratic` til at finde de reelle rødder x til en andengradsligning, der jo har formen

$$p(x) = ax^2 + bx + c = 0$$

for givne reelle tal a, b og c, hvor $a \neq 0$. Der er som bekendt nul, én eller to reelle rødder, dvs. x for hvilke $p(x) = 0$, afhængig af om diskriminanten $d = b^2 - 4ac$ er negativ, nul, eller positiv. Når d er positiv er de to rødder:

$$x_1 = \frac{-b + \sqrt{d}}{2a} \quad \text{og} \quad x_2 = \frac{-b - \sqrt{d}}{2a}$$

og når d er nul, er der én rod:

$$x_1 = \frac{-b}{2a}$$

2.2 Problemanalyse

Med ovenstående beskrivelse kan det se ud som om der slet ikke er nogen problemer at diskutere: Det hele er jo forklaret. I hvert fald virker det oplagt at metode `solveQuadratic` tager tre argumenter `a`, `b` og `c` af type `double`.

Men der er i hvert fald følgende to problemer som skal løses før man kan skrive den ønskede metode:

(A) Hvordan skal metoden præsentere eller returnere resultaterne af ligningsløsningen? Der kan jo være nul, én eller flere rødder; hvordan fortæller metoden både *hvor mange* og *hvilke* rødder der er?

(B) Hvordan skal fejlsituationer håndteres og rapporteres? I problemstillingen forudsættes $a = 0$, for ellers er der slet ikke tale om en andengradsligning, og ovenstående formler duer ikke. Det skal altså overvejes om der skal meldes fejl hvis $a = 0$.

2.2.1 Delproblem A: Hvordan skal metoden levere resultaterne

Der er i hvert fald følgende fem mulige løsninger:

(A1) Metoden kan simpelthen udskrive de nul, én eller to rødder til konsollen, dvs. standard output:

```
public static void solveQuadratic(double a, double b, double c) {
  double d = ...;
  if (d < 0)
    System.out.println("No roots");
  else ...
}
```

Det er nemt at programmere men giver ikke andre dele af programmet mulighed for at bruge rødderne. Og det rejser nye spørgsmål om sprog ("No roots" eller "Ingen rødder"?) samt om antal decimaler i rødderne der udskrives, osv.

Det er altså bedre hvis metoden returnerer rødderne i stedet for at skrive dem ud. Men der skal kunne returneres nul, én eller to rødder.

(A2) Metoden kan returnere et array, af type `double[]`, der indeholder alle rødderne. Arrayets længde er 0, 1 eller 2 afhængig af antallet af rødder.

```
public static double[] solveQuadratic(double a, double b, double c) {
  double d = ...;
  if (d < 0)
    return new double[0];
  else ...
}
```

Fordelen ved denne løsning er at den er ret enkel at programmere. Ulemperne er at brugere af metoden ikke nødvendigvis er klar over at arrayets længde kan variere fra gang til gang, og at hver eneste kald til `solve` bør allokere et nyt array. En alternativ løsning (A2a) er at `solve` genbruger resultat-arrayet, men det er yderst risikabelt, og kan ikke anbefales, fordi en tidligere kalder kan beholde en reference til arrayet, som en nyt kald til `solve` derefter vil opdatere.

(A3) Vi kan definere et lille klassehierarki med en abstrakt superklasse Roots og tre konkrete subklasser NoRoots, OneRoot og TwoRoots, og lade metoden returnere et objekt af en af de tre subklasser:

```
public static Roots solveQuadratic(double a, double b, double c) {
  double d = ...;
  if (d < 0)
    return new NoRoots();
  else ...
}
```

Her ville klasse OneRoot have en get-accessor `getRoot1()` af type `double`, til at hente roden, og tilsvarende for klasse TwoRoots. Fordelen er at denne løsning er robust og ganske objektorienteret. Der er dog samme ulempe som for (A2), nemlig at et objekt skal allokeres for hvert kald til `solveQuadratic`. Desuden kan det virke noget voldsomt at definere fire klasser for at løse en andengradsligning.

(A4) Metoden kan returnere en iterator af type `Iterable<Double>` (i Java) eller IEnumerable<double> (i C#). Denne iterator vil så producere 0, 1 eller 2 rødder, som kan hentes med en for-each løkke.

```
public static Iterable<Double> solveQuadratic(double a, double b, double c) {
  double d = ...;
  if (d < 0)
    return new Iterable<Double>() { ... };
  else ...
}
```

Fordelen ved denne løsning er den er meget generel og også virker for problemer der har 17 mulige resultater, hvor alternativ (A3) ville blive alt for omstændelig. Ulempen er at det er besværligt at programmere korrekt i Java (i C# er det meget nemmere med `yield return`), og at tids- og pladsforbruget til konstruktion af nye objekter er endnu højere end i (A2) og (A3).

(A5) Endelig kunne man i stedet for en metode `solveQuadratic` implementere en klasse SolveQuadratic, hvis konstruktor tager argumenter `a`, `b` og `c`, og som har get-accessors der fortæller hvilke hvor mange og hvilke rødder der er, i denne stil:

```
public class SolveQuadratic {
  ...
  public SolveQuadratic(double a, double b, double c) { ... }
  public int getRootCount() { ... }
  public double getRoot1() { ... }
  public double getRoot2() { ... }
}
```

Fordelen ved denne løsning er at man kan spørge SolveQuadratic objektet om antal rødder og om selve rødderne lige så mange gange man vil, og at man kan tilføje yderligere metoder, fx til at beregne $p(x)$ for et givet x, hvilket kan bruges til at undersøge hvor præcis en løsning er. Ulempen er som før at der allokeres et nyt objekt hver gang en ligning skal løses, og at objektet først skal oprettes, derefter spørges om antal rødder og endelig om de enkelte rødder.

Sammenfattende vælger vi array-løsningen (A2) som et kompromis mellem gennemskuelighed, generalitet, og enkelhed.

2.2.2 Delproblem B: Hvordan skal metoden håndtere fejl

Der er mindst fem måder metoden kan reagere hvis $a = 0$:

(B1) Skriv en besked på konsollen og stop programmet. Duer naturligvis ikke hvis det skal indgå i et større program.

(B2) Ignorér problemet, det må være den kaldende metodes opgave at tjekke at argumenterne er korrekt. Fordelen er at dette ikke kræver nogen handling i `solveQuadratic`. Ulempen er at resultaterne fra metoden uventet kan være de særlige `double`-værdier plus eller minus uendelig, eller `NaN` (not a number).

(B3) Kast en exception. Fordelen er at det passer bedre til Javas generelle filosofi og er meget mere robust end (B2).

(B4) Returnér `null` i stedet for et array. Ulempen ved dette er at den kaldende metode både skal tjekke om det returnerede array er `null`, og hvilken længde det i så fald har. Og det første tjek er nemt at glemme.

(B5) Når $a = 0$ så har vi en førstegradsligning, og man kunne jo bare løse den. Men hvis også $b = c = 0$ så er der uendelig mange løsninger, og der bliver brug for en måde at meddele dét på, hvilket unødigt komplicerer sagerne hvis man faktisk kun er interesseret i at løse egentlige andengradsligninger hvor $a \neq 0$ som angivet i problemstillingen i afsnit 2.1.

Sammenfattende vælger vi løsning (B3) fordi den er enkel og robust.

3 Eksempel 2 på problemanalyse: Kald af ark-definerede funktioner

Dette er et autentisk eksempel fra afsnit 10.1 i en rapport fra et forskningsprojekt om regnearksteknologi. Det er mere kortfattet og realistisk end det foregående eksempel. Det er selvfølgelig næsten ubegribeligt for udenforstående, men budskabet her er jo ikke det tekniske indhold. Budskabet er at man, uanset hvor kompliceret sammenhængen er og uanset hvor erfaren en programmør man er, kan diskutere alternative løsninger inden man har skrevet én linje kode — og derved spare en masse tid og bøvl.

3.1 Problemstilling og baggrund

A sheet-defined function should be able to call other sheet-defined functions, and even itself.

3.2 Problemanalyse

There are at least four different ways to compile a call to a sheet-defined function `sdf`.

- The first approach is to generate code that looks up the sheet-defined function by its name `sdf` in a table to get the delegate representing it, and then calls that delegate. This immediately allows sheet-defined functions to be recursive and mutually recursive, but incurs the cost of the table lookup at each invocation, which is slower than a call to a built-in function from an interpreted formula (where the function name has been replaced by a delegate reference before evaluation).
- The second approach is to retrieve, at generation time, the MethodInfo object corresponding to the delegate compiled for `sdf`, and then generate a bytecode call straight to that MethodInfo object. This avoids the table lookup at call time but precludes recursive and mutually recursive sheet-defined functions, because a sheet-defined function could not be called before it has been defined. Also, if `sdf` were modified and recompiled, all sheet-defined functions calling it would have to be recompiled as well.
- A third and intermediate approach is to maintain a map from names of sheet-defined functions to indexes $0, 1, 2, \ldots$ of sheet-defined functions, and a static array `sdfDelegates` of type `DynamicMethod[]` that maps an index to the delegate generated for that sheet-defined function. A call to the sheet-defined function with index `i` then gets compiled to an array access followed by an invocation `sdfDelegates[i]()`. Whenever the sheet-defined function with index `i` has been recompiled, the entry at `sdfDelegates[i]` must be updated with the new delegate. This permits recursive and mutually recursive sheet-defined functions, and replaces a hash table lookup by a fast array access.
- A fourth approach would be to store a direct reference to the delegate within the sheet-defined function object to be called. To keep this reference up to date, an event, called "compiled" is added to every sheet-defined function, and listeners are added to this event, causing it to update all those delegate fields upon (re)compilation. This would save one array indexing per call and hence be even faster than approach three, but is more likely to go wrong.

For now, approach number three seems to be the simplest. It has been implemented in classes SdfManager and SdfInfo in file `SdfManager.cs`. In summary, an n-argument sheet-defined function `sdf` can be called from another sheet-defined function by the following simple code sequence:

- evaluate and push the n arguments on the stack;
- push integer `i` on the stack, where `i` is the index of `sdf` in the `sdfDelegates` array;
- call static method `SdfManager.SdfMethod`n where n is the number of arguments to `sdf`.