Dokumentation av Källkod

Mikael Svahnberg*

2024-01-12

1 Introduktion

- Dokumentera din kod
- Självdokumenterande kod
- Struktur som dokumentation
- Dokumentera din arbetsprocess

Boktips:

- R. Martin *Clean Code – A Handbook of Agile Software Craftmanship*, Pearson, 2008.
- S. McConnell *Code Complete: A Practical Handbook of Software Construction*, 2nd Edition, Microsoft Press, 2004.

2 Syftet med Dokumentation av Källkod

- Kommunicera med dig själv
  - Vad behöver jag göra i den här metoden för att komma från input till rätt output?
  - Vad heter den här variablen? metoden? Var skrev jag den?
- Kommunicera med ditt framtida själv
  - Hur tänkte jag här?
  - Hur lägger jag till den här funktionaliteten?
  - Har jag redan skrivit detta?
- Kommunicera inom ditt team
  - Hur tänkte `NN` här?
  - Hur lägger jag till den här funktionaliteten?
  - Hur använder man paketet/klassen/metoden?

*Mikael.Svahnberg@bth.se

- Kommunicera med andra utvecklingsteam
  - Hur använder man paketet/klassen/metoden?
- Kommunicera med din chef
  - Vi är till `x %` klara med den här metoden
  - Alla enhetstester är gröna, alltså är koden tillräckligt testad.

3 Dokumentation av Kod

- Hur man *använder* klassen/metoden, inte hur den är implementerad.
- "Kontrakt" mellan utvecklaren och användaren.
- Verktygsstöd:
  - javadoc
  - doxygen
  -
- Ibland stöd redan i programspråket
  - eg. Clojure/Lisp

4 Råd för Dokumentation

- Kortfattat vad metoden/klassen gör
- Inkludera *Exempel*
- Edge Cases
- Vilka alternativ som finns. Särskilt om man förbättrar något som egentligen finns inbyggt i språket.
- För klasser:
  - Författare, version, mm.
  - Om nödvändigt, en lite längre text.
- För metoder:
  - Paramterar, och hur man använder dem.
  - Returvärden
  - Referenser till annan relevant dokumentation

5 JavaDoc Exempel

```
/**
 * Represents a Circle that is defined by a framing rectangle.
 *
 * This class acts as a superclass of all shapes vaguely round.
 *
 * @author Mikael Svahnberg
 * @version 1.0
 */
public class Circle {

/**
 * Sets the location of the Shape
 *
 * @param xPosition the X coordinate of the upper-left corner of the framing rectangle.
 * @param yPosition the Y coordinate of the upper-left corner of the framing rectangle.
 *
 * @return true if it was possible to move the Shape.
 *
 * @see java.awt.geom.Ellipse2D
 */
public boolean move(int xPosition, int yPosition) {
}

}
```

```
mkdir -p DocExample
javadoc -d DocExample Circle.java
xdg-open DocExample/index.html
```

6 Exempel på inbyggd dokumentation

```
(defun emacs-version (&optional here)
  "Display the version of Emacs that is running in this session.
With a prefix argument, insert the Emacs version string at point
instead of displaying it.
If called from Lisp, by default return the version string; but
if the optional argument HERE is non-nil, insert the string at
point instead.

Don't use this function in programs to choose actions according
to the system configuration; look at `system-configuration' instead."
  (interactive "P")
  (let (( ; ...
         ))))
```

7 Att använda sig av Kod-dokumentation

- Finns mycket online, men jag rekommenderar att ha dokumentationen lokalt.
  - Alltid tillgängligt
  - Även på resande fot (30 000 fot upp i luften till exempel)
  - Även bakom företags brandväggar
- Lär dig hur dokumentationen är strukturerad så du vet vad du kan hitta.
  - Bli snabb på att hitta rätt → bli snabb på att återvända till koden.

(Visa exempel från:)

- JavaDoc, (ex. något från Java Standard Library)
- Info, (ex. `info info`)
- man (ex. `man fprint`)

8 Kritik mot JavaDoc och liknande ramverk

- Bra för ett publikt API som skall användas av andra
- Tar tid att skriva och tillför lite för interna klasser/metoder som bara skall ses internt i projektet
- Kostar att hålla uppdaterade tillsammans med koden
- Om man skriver javadoc-kommentarer så förväntas man vara konsekvent och *alltid* skriva dem.
  - Även när metoden, parametrarna, och/eller returvärdet är uppenbara.
- Använd hellre (säger kritikerna) :
  - *Mindre metoder* med ett små och tydliga ansvarsområden
  - *Rätt namn på metoden* så att man av namnet tydligt ser vad den gör
  - *Rätt namn på parametrar* Tillsammans med metodnamnet ger de ofta tillräckligt med ledtrådar
  - *Undvik överraskningar* Var konsekvent i parameterordning och returvärden

9 Kommentarer i koden

- De finns de (t.ex. Robert "Uncle Bob" Martin) som hävdar att man bör undvika kommentarer i koden.
  - *"kod-kommentarer är ett misslyckande [. . .] att skriva läsbar kod"*
  - Fel använt tillför de ingenting
  - Kommentarer glider ofta "isär" från koden de gäller efter hand som man skriver mer kod.
  - Koden uppdateras, kommentaren står kvar och blir mer och mer fel.

10 Kommentarer som Arbetsredskap

Använd kommentarer som ett *arbetsredskap*

- En skiss över vad du vill göra
- Steg för steg hur algoritmen borde gå till
- en byggnadsställning att skriva koden runt.

När du är färdig tar du bort byggnadsställningen (kommentarerna)

- Om det nu är otydligt vad metoden gör, *Refactor*
  - Rätt namn på klassen/metoden?
  - Rätt namn på variablerna?
  - Bryt isär till fler metoder?
    - Ofta blir var och en av dina första kommentarer en lagom stor metod.
- Om det fortfarande är otydligt, lägg till kommentarer som förklarar *vad du menar*.
  - *Inte* vad kompilatorn gör.

11 Acceptabla Kommentarer

- Copyright-information och vilken licens som koden lanseras med
- Förklara varför du valt en viss algoritm
- Förtydliga en bit krånglig kod som du misslyckas med att skriva enklare
  - t.ex. exempel för ett reguljärt uttryck
  - Specialfall för returvärden som annars skulle kunna överraska
- Varningar
- TODO-kommentarer

12 Dåliga Kommentarer

Obegripliga kanske fanns en tanke bakom dem en gång i tiden?

Överflödiga framgår t.ex. redan av metodnamnet.

Felaktiga stämmer inte med vad koden faktiskt gör.

Obligatoriska oavsett om de tillför något eller inte.

Journalanteckningar syns ändå i konfigurationshanteringsverktyget.

Passivt eller Aktivt Aggressiva skrik åt kaffebryggaren i stället för i kod-kommentarer.

Bortkommenterad kod är i vägen men ingen vågar radera; de bygger upp över tid

Mötesprotokoll som kanske förklarar ett designbeslut men inte tillför koden något.

Information långt bortifrån kommentaren skall handla om vad som är här och nu.

13 Literate Programming

- Vänder ut och in på idén med kommentarer.
- Skriv prosa först, och programkod sedan.
- Tidigare nämnt i kursen:

  > Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. – Donald Knuth, 1084.

- Programmera som om du skriver en lärobok för ditt program.
  - Exempel: `https://howardism.org/Technical/LP/introduction.html`
- Börja med att skriva prosa om vad du vill åstadkomma
- Ge exempel
- Illustrera med design-diagram
- Sist, skriv den faktiska koden.

Bra/Dåligt?

- Det här liknar din ingenjörsdagbok
- Personligen använder jag LP mer för DevOps än för vanlig programmering
  - Explorativt
  - Skriver läroboken för mig själv
  - Lätt att söka och hitta med rubriker som i en vanlig text.
  - Det finns inte alltid en övergripande ordning i vilken saker skall göras.
    - Om det gör det, skriver jag hellre ett script.
  - Det är jag själv som skall manuellt "exekvera" de olika blocken

14 Namn som Dokumentation

- Namnet på en variabel/metod/klass/... skall tillräckligt beskriva vad entiteten gör.
  - Kräver att entiteten har ett litet och väl avgränsat ansvarsområde.
- Det är ok med långa namn: Hellre `elapsedTimeInDays` än `d`
- Kanske i omkastad ordning `daysElapsed`
  - Ifall företagspolicy tillåter det
  - Ifall det blir lättare att använda tab-complete på det ena eller det andra sättet.
  - Ifall du är konsekvent och *alltid* döper dina variabler på det viset.
- Använd konstanter i stället för magiska tal
  - `if (200 == x) {}`
  - blir till `if (STATUS_OK == returnStatus) {}`
- Använd uttalbara namn
  - du läser koden som text och skall också kunna prata om den
  - det är också lättare att komma ihåg
  - de blir indirekt också *sökbara*

15 Namn på Klasser och Metoder

- Namn på klasser är troligen *Substantiv*
- Namn på metoder innehåller troligen *Verb*
  - Kanske kombinerat med ett substantiv
  - ... om inte det är uppenbart att det är på klassen den verkar

```
Figure currentFigure = getSelectedFigure();

currentFigure.setPosition(100, 100);
currentFigure.show();
```

16 Mindre och Enklare Metoder

- Bryt isär flerstegsmetoder till separata metoder istället.
  - Var och en med ett tydligt namn.
- Så få parametrar som möjligt (Uncle Bob: *Helst inga parametrar alls. Någonsin!*)
- Undvik sidoeffekter (föredra rena matematiska funktioner); det blir lättare att förstå
- Undvik att förändra inparametrarna – föredra `return` i stället.
  - Gör kanske något på objektet i stället: `fancyObject.applySomeDecoration()`

17 Metodanrop som dokumentation

- Algoritmer kan dokumenteras med hjälp av metodanrop. Exempel:

```
private int partition(int[] arr, int leftIdx, int rightIdx) {
  int pivot = arr[Math.floorDiv((leftIdx + rightIdx), 2)]; // Pick an element somewhere in

  while (leftIdx <= rightIdx) {
    // "sneak up" on the middle from both ends; keep going
    // as long as the elements are already smaller/bigger than the pivot
    while (arr[leftIdx] < pivot) { leftIdx++; }
    while (arr[rightIdx] > pivot) { rightIdx--; }

    // If they're not, and we still have elements to go
    // swap the two values that are out of order and keep going
    if (leftIdx <= rightIdx) {
      int tmp = arr[leftIdx];
      arr[leftIdx] = arr[rightIdx];
      arr[rightIdx] = tmp;

      leftIdx++;
      rightIdx--;
    }
  }

  return leftIdx; // This is the final pivot point
}
```

Kan förtydligas till:

```
private int partition(int[] inputArray, int leftIndex, int rightIndex) {
  int pivotValue = getPivotValue(inputArray, leftIndex, rightIndex);

  while (!isSortingDone(leftIndex, rightIndex)) {
    leftIndex = findNextElement(inputArray, leftIndex, GREATER_THAN, pivotValue);
    rightIndex = findNextElement(inputArray, rightIndex, SMALLER_THAN, pivotValue);

    if (!isSortingDone(leftIndex, rightIndex)) {
      swapValues(inputArray, leftIndex, rightIndex);
      leftIndex++; // No need to look at these values again
      rightIndex--;
    }
  }

  return leftIndex; // This is the point in the array where we were done sorting.
}
```

Det går att göra mer med den här koden. **Diskutera**

- parametrar
- `isSortingDone()`
- `(maybe-swap left right)`

18 Struktur är också Dokumentation

- Paket och Moduler syns ofta i filsystemet som kataloger
  - Lätt att navigera till rätt klass/paket
  - Paketnamnet ger information om sammanhanget
  - `Cowboy.Gun.draw()` är något helt annat än `Cliparts.Gun.draw()`
- Namespace ger möjlighet att förenkla namnen på klasser och metoder
  - Jämför med emacs-lisp, där allt ligger i ett globalt namespace.
    - Vanlig struktur på funktionsnamn `(author-package-filename-method)`

```
$ tree cljdetector/
cljdetector/
 CHANGELOG.md
 doc
    intro.md
 Dockerfile
 LICENSE
 project.clj
 README.md
 resources
 src
    cljdetector
        core.clj
        process
           expander.clj
           source_processor.clj
        storage
            storage.clj
 target
    [...]
 test
     cljdetector
        core_test.clj
     data
         A.java
         B.java

17 directories, 15 files
```

19 Konfigurationshantering är också Dokumentation

- Radera kod i stället för att kommentera bort den
  - Det är ok att kommentera bort kod under tiden du arbetar *men* ta bort den innan commit.

- När du raderat kod: *commit*
- När du committar:
  - Beskriv i meddelandet vad du gjort
  - Länka till Issues (kan stänga en issue automatiskt från commiten *"Fixes issue #222"*)

20 Dokumentera Projektet: README

- Använd *minst* en `README` - fil för att dokumentera projektet
- Det finns verktyg för att skapa en grundläggande `README`
- `README.md` ligger ofta i root-katalogen på ett projekt
- Innehåll
  - Projektnamn
  - Beskrivning
    - Vad handlar projektet om
    - Skärmdump eller annan illustration
    - Vilket problem löser det
    - Vilka teknologier har du använt
    - Särskilda utmaningar du har eller hoppas ta hand om.
  - Innehållsförteckning (om lång README)
  - Instruktioner (kanske separata sidor)
    - Instruktioner för att Installera och Köra Projektet
    - Instruktioner för att använda projektet
    - Instruktioner för hur man kan anpassa projektet
    - Instruktioner för hur man testar projektet
  - Credits, information om utvecklingsteamet, andra källor, mm.
  - Instruktioner om hur läsaren kan bidra
  - Licens

21 Dokumentera Buggar

- Idag använder man ofta en *Issue Tracker* integrerat i arbetsflödet
  - Hitta ett problem eller förbättringsförslag → Skapa en issue
  - Diskutera issuen
  - Klassificera Issues (e.g. Bug, Förbättring, Dokumentation, Fråga, . . .)
  - Prioritera Issues, planera när du skall arbeta med dem
  - Tilldela till utvecklare
  - Lös med en `commit`

- Olika projekt har olika förväntningar, men en bra issue innehåller oftast:
  - Kort men innehållsrik titel
  - Förbättringsförslag? → Beskrivande Text
  - Bug?
    * Version av Programvaran och kringliggande teknologier (e.g. version på programtolken, databas, ...)
    * Steg för att provocera fram buggen (minsta möjliga / enklaste stegen för att provocera)
    * Förväntat Resultat
    * Faktiskt Resultat (med log-utskrifter och skärmdumpar)
    * Övrig information

22 Dokumentera Arbetsprocessen

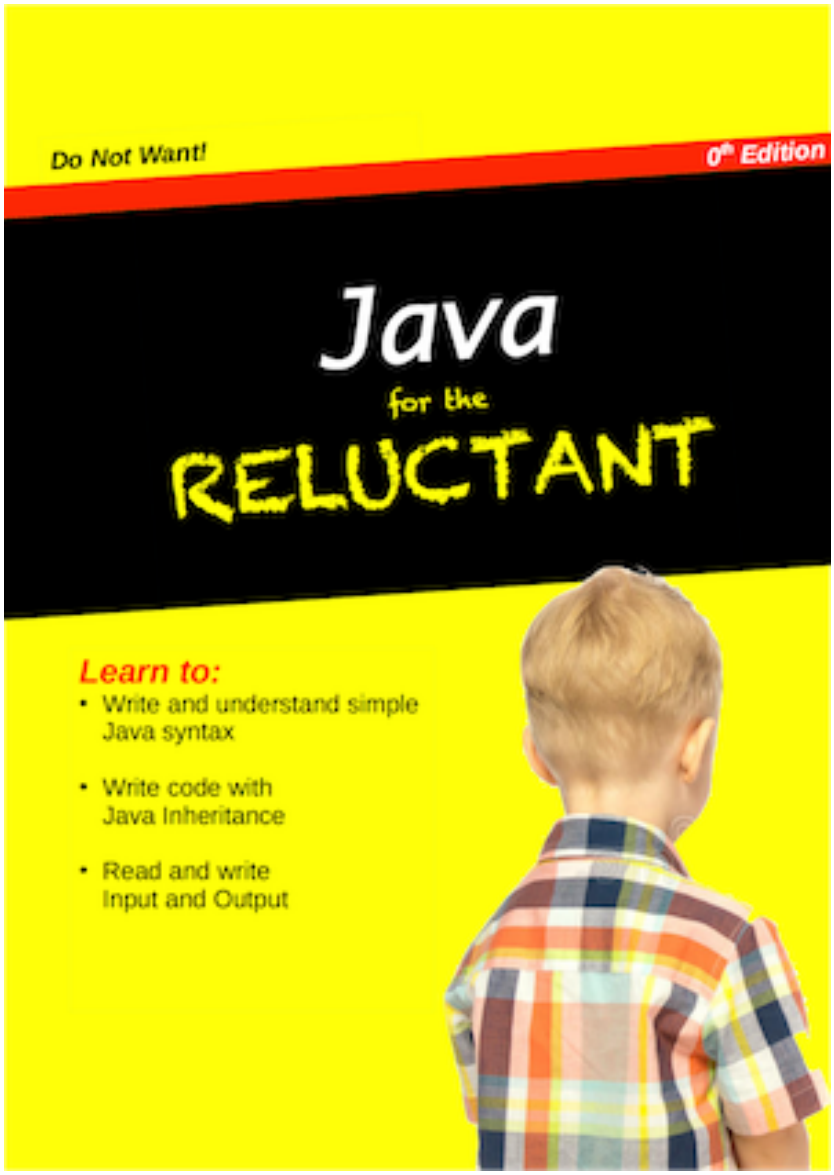
- En arkitektur / design består främst av de *beslut* som är fattade.
- Undvik att behöva återskapa dessa:
  - Dokumentera beslut.
  - Dokumentera design och arkitektur.
- När och Hur skapar man en ny branch?
- När och Hur skapar man en testmiljö?
- Vem och Hur lanserar man produkten (deployment)?
- Lika viktigt är dina egna lärdomar. → *ingenjörsdagboken* .

23 Sammanfattning

- Dokumentation på *rätt* sätt förklarar där det behövs
- Dokumentation på *fel* sätt är i vägen, oftast felaktig, och kostar arbetstid.
- Dokumentation utanför koden
- Dokumentation från koden (t.ex. javadoc)
- Dokumentation inuti koden
- Självdokumenterande kod.

24 Nästa Föreläsning

- Ett exempel på ett Kompilerat programspråk: Java
  - Kompileringscykel
  - Klasser, Objekt, Attribut
  - Vilkor, Iterationer
  - Paket
- Titta gärna på Youtube-serien "Java for the Reluctant"
  - `https://www.youtube.com/playlist?list=PLRyU9jMTRIX7QUrxZF9HNLbdXtiv6F9MT`
  - `https://github.com/mickesv/Reluctant-Java.git`



25 Övning: Dokumentation

26 Introduktion till JavaPonies

- *Desktop Ponies* är en urgammal mono-applikation (liknar Visual Basic) som låter My Little Ponies springa runt på skärmen.
- *Java Ponies* är "min" version av detta program
  - `https://codeberg.org/mickesv/JavaPonies.git`
  - Varning:
    - Det är långt ifrån färdigt
    - Det är inte fulständigt dokumenterat
    - Det är långsamt och säkert buggigt
  - Men:
    - PONIES

`./fly_rainbow_right.gif`

27 Kom igång med övningen

1. Klona projektet till din dator: `https://codeberg.org/mickesv/JavaPonies.git`
2. Öppna projektet i din IDE och studera programmet så att du förstår vad det gör.
   - `src/JavaPonies.java` startar programmet
   - `src/model/Pony.java` Implementerar en klass som instantieras för varje ponny
   - `src/model/PonyBehaviour.java` representerar ett enskilt beteende som en ponny kan ha
   - `src/view/PonyWindow.java` Sköter visning och uppdatering av en viss ponny som är aktiv på skärmen.

28 Skapa dokumentationen

- Från en terminal: `javadoc src/*.java src/model/*.java src/view/*.java -d doc`
- Från IntelliJ: `Tools/Generate JavaDoc`, fyll i att dokumentationen skall hamna i katalogen `doc` (Resultatet öppnas i din webläsare)

Inspektera dokumentationen:

- Jämför med vad du ser i java-filerna
- Vad finns med? Vad finns inte med?
- När du skapade dokumentationen fick du många varningar. Vad beror de på?

29 Uppdatera JavaDoc

Filen `src/model/PonyBehaviour.java` saknar JavaDoc-kommentarer.

1. Skriv dessa kommentarer så att du inte längre får några javadoc-varningar från `PonyBehaviour.java`.
2. Ökade detta användbarheten av dokumentationen? Varför / Varför inte?
3. Ökade detta läsbarheten av koden? Varför/Varför inte?

30 Skapa en Issue

Notera:

- Om du har ett konto på Codeberg.org kan du skapa en issue direkt mot JavaPonies-projektet
- Om du inte har eller inte vill ha ett konto kan du skriva din issue direkt i en textfil.

Att Göra:

1. Hitta någonting att åtgärda i projektet. Det kan vara en:
    - Bug – något som inte fungerar som förväntat
    - Enhancement – en ny feature
2. Skriv din issue. Den skall innehålla:
    - Kort men innehållsrik titel
    - Beskrivande text
    - Steg för att provocera fram buggen, eller steg till där förbättringsförslaget skulle kunna vara lämpligt
    - Förväntat resultat
    - Faktiskt resultat
    - Övrig information (om relevant)

31 Förbättra en metod

1. Välj en metod eller attribyt som du tycker är otydlig och genomför en *Refactoring* så att den blir tydligare.
    - IntelliJ har en hel meny för Refactor; utforska den för att se vad som finns där och hur det fungerar

Fundera på:

- Vilket stöd har du av din IDE för Refactoring?
- Hjälper detta stödet?
- Hur vet du vad som är en bra refactoring?