

F5

Enkel Design, Refaktorisering

EDAF45

Programvaruutveckling i grupp – Projekt
Görel Hedin, Boris Magnusson, Datavetenskap, LTH

XP:s Deltekniker (Practices)

1. Planering Planeringsspelet Regelbundna releaser Hållbart tempo Kund i teamet	2. Utveckling Test-driven utveckling (TDD) Parprogrammering Kollektivt ägande Kontinuerlig integration
3. Kodning och Design Enkel design Refaktorisering Kodningsstandard Gemensam vokabulär	4. Dessutom Gemensamt utvecklingsrum Nollte iterationen Spikes

Refaktorisering

- Omstrukturering av koden utan att ändra det yttre beteendet

Enkel Design

Simple Design

- Vår kod har enkel design om
 - den är tydlig, lättbegriplig
 - ingen duplicerad kod
 - all komplexitet skall vara motiverad av dagens behov – testfallen

Exempel på icke-enkel design

- krånglig kod, obegripliga namn
- copy-paste kod
- klass och metodskelett som inte används än
- patterns som används “i onödan”, innan de verkligen behövs

Varför är Enkel Design viktigt?

- **XP's motivering**
 - en komplicerad initial design kan vara spild tid – projektet ändrar riktning
 - en komplicerad slutlig design blir bättre om man gör den i många små steg
- **Bygger på att vi kan/vågar ändra designen, vilket underlättas av**
 - att koden är ren och tydlig
 - att koden täcks av testfall som fångar fel vi råkar införa vid ändringar
 - att vi systematiskt kan refaktorisera koden, helst med verktyg

Motsats till enkel design

- “Big design upfront” (BUF)
 - Detaljerad komplex design innan vi börjar implementera
 - Implementera alla klass och methodskelett i UML-verktyg innan man börjar implementera koden

I XP vill man istället...

- Initial design på whiteboard. Diskutera flera olika möjligheter.
- Ingen fullständig detaljerad design på detta stadium
- Implementera i baby steps, få feedback på vilken design som fungerar i praktiken
- Lägg till designkomplexitet, t.ex. olika patterns, efter behov under implementationen, som del av de stories som behöver denna komplexitet.
- Se till att kontinuerligt diskutera designen så att alla i teamet är med på hur den nuvarande designen ser ut

Enkel Design:

tydlig, lättbegriplig kod

- Vi skall kunna förstå koden utan att behöva dechiffrera den.
- T.ex.:
 - goda val av namn
 - namnge värden och beräkningar (som variabler och metoder)
 - i stället för att bara koda algoritmen direkt
 - varje metod skall vara så liten och enkel att man lätt kan förstå vad som händer i den, och så att den är enkel att namnge
 - varje variabel skall användas till ett enkelt väldefinierat ändamål, så att den är enkel att namnge
 - dito för klass...
- och omvänt...
 - ingen “dålig lukt” i koden

Exempel på kod som behöver dechiffreras

```
...  
String s = ...;  
int count = 0;  
char c;  
char prevC = 'x';  
for (int k=0; k< s.length(); k++) {  
    c = s.charAt(k);  
    if (c != ' ' && prevC == ' ') count++;  
    prevC = c;  
}  
if (c == ' ') count--;  
count++;  
...
```

- Börja med Extract Method -- extrahera ett stycke kod till en metod.
- Ge den ett lämpligt namn - **ja vad gör koden ???**

Refaktoriserad kod

```
String s = ...;
int count = numberOfWordsIn(s);
...

int numberOfWordsIn(String s) {
    int count = 0;
    char c;
    char prevC = 'x';
    for (int k=0; k< s.length(); k++) {
        c = s.charAt(k);
        if (c != ' ' && prevC == ' ') count++;
        prevC = c;
    }
    if (c == ' ') count--;
    return count++;
}
```

- Designen kan kanske förenklas ytterligare
 - ... men först Testfall

Testfall - förväntat utfall

- `assertEquals("Test1",0,numbersOfWordsIn(" "));`

0	" "
0	" _ "
1	"A"
1	"_A"
1	"A_ "
1	"_A_ _ "
2	"A_B"
2	"_A_B"
2	"A_B_ "
2	"ABC_DEF"
7	"A_B_C_D_E_F"

Testfall - verkligt utfall

1	0	" "
	0	" "
	1	"A"
	1	" A"
0	1	"A_"
0	1	"_A_"
	2	"A_B"
3	2	"_A_B"
	2	"A_B_"
	2	"ABC_DEF"
	7	"A_B_C_D_E_F"

Refaktorerad kod

```
int numberOfWordsIn(String s) {  
    int wordCount = 0;  
    char prevC = ' '  
    char newC; /* Recognize shift from space to non-sp */  
    for (int k=0; k< s.length(); k++) {  
        newC = s.charAt(k);  
        if (prevC == ' ' && newC != ' ' )  
            wordCount++;  
        prevC = newC;  
    }  
    return wordCount;  
}
```

- Och nu kör alla testfallen !

Man kan gå vidare ...

```
boolean startOfWord(char c1, char c2) {  
    return (c1 == ' ' && c2 != ' ');  
}
```

```
int numberOfWordsIn(String s) {  
    int wordCount = 0;  
    char prevC = ' '  
    char newC;  
    for (int k=0; k< s.length(); k++) {  
        newC = s.charAt(k);  
        if( startOfWord(prevC,newC))  
            wordCount++;  
        prevC = newC;  
    }  
    return wordCount;  
}
```

- Kör testfall efter varje förändring!

Tydlig lättbegriplig kod

- T.ex.
 - Sätt bra namn på varje beräkning och värde
 - Namnen ska uttrycka “vad”, och inte “hur”.
- XP Slogans
 - “Express every idea”
 - “Express intention in the code, rather than algorithm”

Enkel Design:

Ingen duplicerad kod

- Duplicerad kod uppkommer väldigt ofta
 - copy-paste är ett enkelt effektivt sätt att lösa problem
- Problem med duplicerad kod
 - Programmet kan bli onödigt stort och tar onödigt lång tid att läsa
 - Lätt att glömma uppdatera alla kodavsnitten vid en förändring
 - Det verkar finnas en återkommande “idé” som skulle kunna extraheras och göras explicit.

Exempel på duplicerad kod

```
class Stack {  
    void method push(Object o) {  
        if (Tracing.on && Tracing.traces(this))  
            System.out.println("Entering method Stack.push");  
        ...  
    }  
  
    Object method pop(Object o) {  
        if (Tracing.on && Tracing.traces(this))  
            System.out.println("Entering method Stack.pop");  
        ...  
    }  
}
```

- Refaktorisera!
 - Använd “Extract Method” för att göra om den duplicerade koden till en metod, och
 - “Move Method” för att flytta den till en bättre plats.

Refaktoriserad kod

```
class Stack {  
    void method push(Object o) {  
        Tracing.traceEnterMethod(this, "Stack.push");  
        ...  
    }  
  
    Object method pop(Object o) {  
        Tracing.traceEnterMethod(this, "Stack.pop");  
        ...  
    }  
}  
  
class Tracing {  
    void traceEnterMethod(Object o, String methodName){  
        if (on && traces(o))  
            System.out.println("Entering method "+methodName);  
    }  
}
```

- Designen kan kanske förenklas ytterligare...

Duplicerad kod

- Om liknande kod förekommer på flera ställen
 - försök hitta vad det är för “idé” som inte är klart uttryckt
 - försök faktorisera ut den duplicerade koden, exempelvis till en ny metod
 - ofta blir koden både klarare och enklare
- XP slogan
 - “once and only once”

Enkel Design:

Ingen onödig komplexitet

- Fokusera på dagens behov
 - dagens behov definieras av testfallen
 - all komplexitet skall vara motiverad av testfallen
 - vi vidareutvecklar designen efter hand som behov uppstår
- Om vi tror att vi kommer att behöva en mer komplex design senare...
 - vänta med att införa den tills vi verkligen behöver den
 - genom att ha gjort en enklare design först kommer vi att lättare att kunna skapa en bra komplex design

Exempel på design som vidareutvecklas

Banksystem

- Vi utvecklar ett banksystem för en lokal bank i Lund, och behöver en klass för att hantera pengar.
- Primärt krav: att kunna representera pengar som värden som kan adderas, subtraheras, beräknas ränta på, avrundas till närmsta krona, etc.
- Vi inser att vi med tiden kommer att behöva hantera även andra valutor, t.ex. Euro, men detta är inget systemet behöver kunna idag.
- Vi designar för dagens behov – svenska kronor...

Initial design och implementation:

A blue rectangular button with the word "Kronor" in white text.

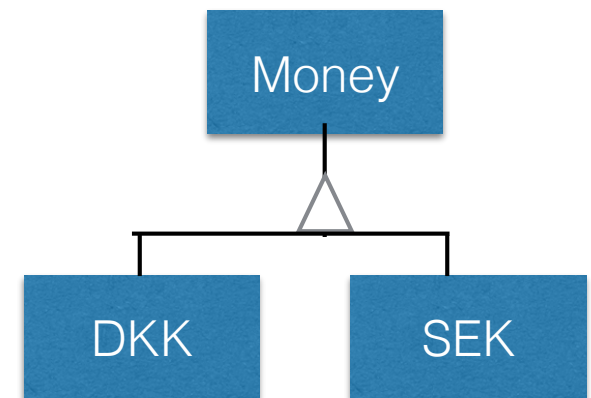
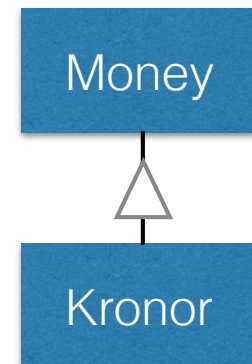
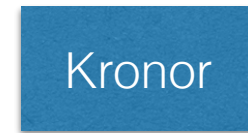
Kronor

Nytt krav

- Vissa företag i regionen vill kunna betala ut löner i andra valutor som Euro och danska kronor.
- Hur skall vi vidareutveckla designen?
- Möjliga strategier
 - Designa i förväg (inte XP): Tänk igenom alla tänkbara problem och ta fram ett komplett UML-diagram över den slutliga designen
 - Designa efter hand (XP): Tag ett litet problem i taget. Skriv testfall. Implementera. Refaktorisera till Enkel Design.

Nytt testfall

- Refaktorisera:
 - Extrahera ny klass Money från Kronor för att göra det enklare att lägga till DKK. Testfall kör fortfarande.
- **Skriv nytt testfall** för ytterligare en valuta, danska kronor.
- Lägg till ny klass DKK
- Refaktorisera:
 - Byt namn från Kronor till SEK för att få mer konsistent namngivning
- **Är detta Enkel Design?**

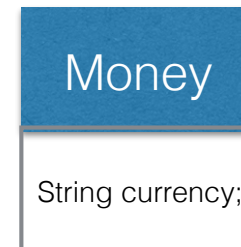


Vi har duplicerad kod

- Subklassernas metoder är mycket lika – de skiljer sig bara åt i den typ de hanterar.
 - Move metod ...
 - All kod hamnar i Money!
- Alternativ enklare design: **parametrisera** Money med en valuta i stället för att ha **subklasser** till Money

Resultande design

- Designen kommer att fortsätta att utvecklas senare... t.ex.:
 - Representera valuta med klass `Currency` i stället för sträng
 - Möjlighet att hantera samling av pengar av olika valutor med hjälp av Composite-mönstret
- Med gamla Brittiska pengar (Pund, Shilling, Pence) hade krävt en **subklass** - tur de försvann i tid!



XP slogans om Enkel Design

- “Do the simplest thing that could possibly work”
 - Vi nöjer oss med Kronor - vi behöver inte Money just nu.
- “YAGNI – You aren’t gonna need it”
 - Vi vet ju inte säkert om vi kommer att behöva flera valutor.
- Undvik “design on speculation”
 - Vi designar inte för flera valutor än - det vore att designa på spekulation – vi vet inte om en sådan komplex design kommer att löna sig.
- Undvik “BUF - Big Upfront Design”
 - Det är mycket svårt att förutse vilken slutlig design som kommer att bli bäst – utan att implementera den. Skall vi ha ärvning eller parametrisering, t.ex? Om vi designar medan vi implementerar får vi feedback från koden om vilken design som blir enklast.

Kan en Enkel Design vara komplex?

- Essential vs. Accidental complexity. [Brooks: No Silver Bullet 1987]
- Javisst, om det är ett **komplext problem** vi löser kan designen också behöva vara komplex.
- Designen är Enkel när
 - varje idé är explicit i koden,
 - det inte finns duplicerad kod,
 - all komplexitet i koden är motiverad av testfallen
 - liten andel accidental complexity.

Enkel Design är ett ideal

- Bättre “idéer” kan mogna fram efter hand.
- Programmeringsspråket är inte alltid kraftfullt nog.
- Viktigt att ha en gemensam kodningsstandard med
 - konsekvent namngivning
 - konsekvent användning av “best practices” (patterns, bibliotek, idiom, ...)
- Efter ett antal ändringar upptäcker man plötsligt att man inte längre har tydlig lättbegriplig kod.

Refaktorisera - När?

- När koden ”Luktar illa”
- “Bad Smells in Code” [Fowler et al. Refactoring. Addison-Wesley 1999]

“Bad Smells in Code”

Signalerar att man kanske *inte* har Enkel Design

- **Duplicated Code**
- **Long Method**
- **Large Class**
- Long Parameter List
- Divergent Change
- **Shotgun Surgery**
- **Feature Envy**
- **Data Clumps**
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Refused Bequest
- **Comments**

Duplicate Code

- Det lättaste sättet att programmera något är att kopiera en bit kod som gör ungefär det man vill, och sedan modifiera koden.
- Ur [Fowler, Refactoring]:
 - **Three strikes and you refactor**
 1. The first time you do something, you just do it.
 2. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway.
 3. The third time you do something similar, you refactor.

Long Method

- Ur [Fowler, Refactoring]:
 - A heuristic we follow is that whenever we feel the **need to comment** something, we write a method instead.
 - Such a method contains the code that was commented but is named after the intention of the code rather than how it does it.
 - We may do this on a group of lines or on as little as a single line of code.
 - We do this even if the method call is longer than the code it replaces, provided the method name explains the purpose of the code.
 - The key here is not method length, but the semantic distance between what the method does and how it does it.

Large class

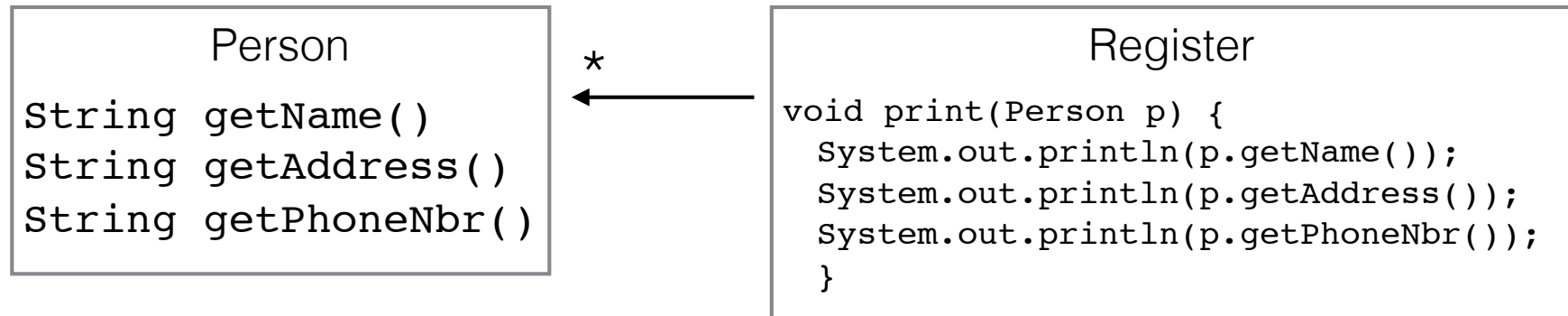
- Ur [Fowler, Refactoring]:
 - When a class is trying to do too much, it often shows up as **too many instance variables**.
 - The usual solution ... is either to
 - Extract Class or
 - Extract Subclass.

Shotgun surgery

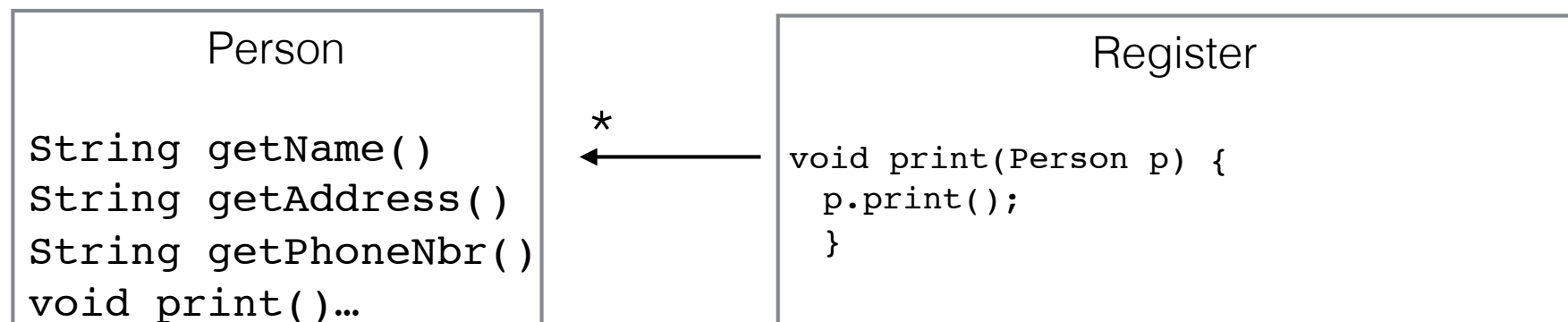
- Tecken: Varje gång man gör en viss slags ändring så behöver man in och **ändra i många olika klasser**
- Lösning: delegera
 - problemet kan ibland lösas genom att flytta de påverkade metoderna till en ny klass med “Move Method”
- Men ibland är programmeringsspråket inte tillräckligt kraftfullt... Mer avancerad programmeringsteknik krävs som
 - introspection/reflection (möjlighet att under exekvering accessa och manipulera en representation av programmet)
(se t.ex. biblioteket `java.lang.reflect`)
 - aspekt-orienterad programming (möjlighet att modularisera ortogonalt mot klasser och metoder)
(se t.ex. <http://www.eclipse.org/aspectj/>)

Feature Envy

- En metod verkar passa bättre i en annan klass än den där den befinner sig (**många punkter** på samma objekt)



- Flytta metoden med "Move Method"



Data Clumps

- Några data-värden förekommer tillsammans på flera platser, som **parametrar** och/eller variabler. De borde egentligen samlas i ett objekt.

```
GraphicalObject
int xpos, ypos, zpos;

void setpos(int x, int y, int z)
int getX()
int getY()
int getZ()
void moveDelta(int dx, int dy, int dz)
void
```

- Position eller Vektor!
- Använd “Extract Class” och “Introduce Parameter Object” för att eliminera data-klumparna.

Comments

Kommentarer luktar egentligen gott.

- Exempel på bra kommentarer:
 - API-kommentarer
 - kommentarer om klasser, paket, etc. deras uppgift, ansvar.
 - kommentarer som klargör varför koden ser ut som den gör
 - kommentarer om oklara saker som behöver arbetas vidare på
- Varningsflaggan gäller **“deodorant”-kommentarer**.
 - Fungerar kommentarerna som “deodorant” för snårig kod?
 - Försvinner behovet av kommentaren om koden refaktoriseras?
- Använd “Extract Method” för att bryta ut snårig del av koden.
- Använd “Rename...” för att göra koden klarare.

Refaktorisering (Refactoring)

- Omstrukturering av koden utan att ändra det yttre beteendet

När gör man refaktorisering?

Hela tiden!

- **Innan en ändring**, för att underlätta ändringen. (Upload så tidigt som möjligt!)
- **Efter en ändring**, för att upprätthålla Enkel Design
- **När man läser kod**, för att förstå den bättre, och gradvis få mer Enkel Design

Exempel på refaktoriseringar

[Fowler]

Composing methods

- Extract/Inline Method
- Introduce Explaining Variable
- Split Temporary Variable
- ...

Moving features between objects

- Move Method
- Extract/Inline Class
- Hide Delegate
- ...

Organizing data

- Replace Data Value with Object
- Encapsulate Field
- Replace Type Code with Subclasses
- ...

Simplifying conditional expressions

- Decompose Conditional
- Introduce Null Object
- ...

forts

Making method calls simpler

- Rename Method
- Add/Remove Parameter
- Separate Query from Modifier
- Introduce Parameter Object
- ...

Dealing with Generalization

- Pull Up/Push Down
Field/Method/Constructor
- Extract subclass/superclass/
interface

- Replace Inheritance with
Delegation

- ...

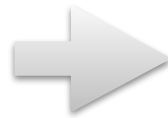
Big refactorings

- Tease Apart Inheritance
- Convert Procedural Design to
Objects
- Separate Domain from
Presentation
- ...

Exempel: Add Parameter

- Vi håller på med en deluppgift (task) och behöver införa en ny parameter till en metod.

```
Person  
void print()
```



```
Person  
void print(Stream s)
```

- I princip trivialt - men ändringar på många ställen.
- Men det är en god idé att göra refaktoriseringen först och kolla att testerna fortfarande fungerar
- Därefter kan man börja implementera den funktionalitet som använder den nya parametern.

“Mekanik” för att undvika många fel på en gång:

- Kontrollera om det finns implementationer av metoden i super- eller subklasser. I så fall skall dessa steg göras för varje implementation.
 - Deklarera en ny metod med den nya parametern. Kopiera det gamla metodinnehållet till den nya metoden.

```
void print(Stream s)
    <kopia>
```
 - Kompilera.
 - Ändra den gamla metoden så att den anropar den nya. Använd t.ex. null som värde på den nya parametern.

```
void print()
    print(null);
```
 - Kompilera och testa.
 - Leta rätt på alla anrop till den gamla metoden och ändra dem så att de anropar den nya metoden. Kompilera och testa efter varje ändring.
 - Ta bort den gamla metoden.
 - Kompilera och testa. Glömt nått ställe?
- Gå vidare med deluppgiften.

```
void print()
    print(null);
```

Länkar om refaktorisering

- Katalog över alla Fowler's refaktoriseringar
<http://www.refactoring.com/catalog/>

Lab 3: Testning och Parprogrammering

- Denna vecka
- Förberedelse - se Föreläsning F04

Labb 4: Refaktorisering

- Utnyttja refaktoriseringsverktyg i integrerad programutvecklingsmiljö (Eclipse)
- Förberedelser
 - F5 Denna föreläsning
 - Labbhandledningen

Sammanfattning

- **Enkel design**
 - tydlig lättläst kod
 - ingen duplicerad kod
 - ingen komplexitet som ej behövs för nuvarande testfall
- **Kodlukter**
- **Refaktorisering**
 - omstrukturering av koden utan att ändra yttre beteendet