PvG

Philip Larsson

11 december 2013

Innehåll

1 Extreme Programming

XP är en agil[1] metod för hur man utvecklar programvara.
Mål med XP är kod med mycket hög kvalité och kod som kan ändras i takt med ändrade krav.

1.1 XP's deltekniker

Kodning och design

- Enkel design
- Refaktorisering
- Kodningsstandard
- Gemensam vokabulär

Utveckling

- Test-driven utveckling (TDD)
- Parprogrammering
- Kollektivt kodägande
- Kontinuelig integration

Planering

- Kund i teamet
- Planeringsspelet
- Regelbundna releaser
- Hållbart tempo

Dessutom

- Gemensamt utvecklingsrum
- Nollte iterationen
- Spikes

1.1.1 XP Practices

Från Artikel av Kent Beck (i häftet).

- Planning game
- Small releases
- Metaphor
- Simple design (no duplicated code, few classes and methods)
- Tests (unit tests)
- Refactoring
- Pair programming

[1] agil: lättrörlig, vig

- Continuous ownership
- Collective ownership
- On-site customer
- 40-hour weeks
- Open workspace
- Just Rules

1.2 Planeringsspelet

Ska svara på frågan ”Vad skall vi utveckla och hur lång tid tar det?”
Kunden skriver användarfall/user stories.
Utvecklaren estimerar tid för varje story.

1.3 Regelbundna releaser

Releaser till kund ska göras ofta.
Detta ger snabb feedback och kunden kan lätt påverka vidareutvecklingen.

1.4 Kund i teamet

En kund bör finnas på plats för att kunna kommunicera med utvecklarna. Personen behöver inte vara beställaren utan kan vara någon som representerar kunden och dess önskemål.

1.5 Gemensam vokabulär

Samma vokabulär mellan kund och utvecklare. Använd gärna metaforer.

1.6 Testning

Testning har en central del i XP.
Enhetstest (unit-tests) för varje klass/modul. Testfall skrivs **innan** motsvarande kod. Skrivs i små iterationer: testa..koda..testa..koda..
Körs gärna automatiskt.

1.7 Kodning och design

1.7.1 Enkel design

- Ingen duplicerad kod.
- Tydlig kod med bra namn.
- Ingen onödig komplexitet.
- Viktigt att programmet är lätt att förstå och ändra i.

Designa för dagens behov, inte för eventuella behov i framtiden.

> ”Don’t design och speculation”

1.7.2 Refaktorisering

Omstrukturering av kod utan att ändra beteendet. Exempel på metoder:

Rename Method: byta namn på metod.

Extract Method: bryt ut kod till egen metod.

Move Method: flytta en metod till en annan klass.

Refaktorisering gör man för att förstå koden bättre. För att lättare kunna införa en ändring eller när koden börjar ”lukta illa”.

1.8 Dokumentation

XP fokuserar inte på dokumentation. Man använder endast enkel och informell dokumentation med enkla beskrivningar. Koden ska vara självförklarande, så god namngivning är ett måste.
Mer noggrann dokumentation kan tas fram i efterhand om kund önskar det.

2 Arkitektur och planering

Mjukvaruarkitektur är den övergripande designen. Traditionell syn på arkitekturen är att den är svår att ändra. Därför planerar man och designar arkitekturen först.
I XP så accepterar man att förändringar är oundvikliga, och gör lite för att förhindra dessa. Istället omfamnar man dem när de sker.

2.1 XP’s grundläggande struktur / arkitektur

Spikes

Är experimentell prototyplösning. En eller två utvecklare skriver kod för att undersöka ett problem. Koden bör inte vara ren och enligt konstens alla regler - koden kommer att kastas sedan.

System Metafor

System metaphor är en liknelse som alla - kund, programmerare och chefer förstår. Används för att beskriva hur systemet fungerar.

Första Iterationen

I den första iterationen så vill man bara ha ett minimalt system som kan köras, med minimal funktionalitet. Man väljer några enkla grundläggande stories som man tror kommer att leda till att man har en grundläggande struktur.

Små releases

Korta releasecykler har flera fördelar. Kunden ser vid varje release hur arbetet framskrider (och vad hon betalar för). Korta releasecykler leder även till feedback som gör att man snabbt kan ändra om man t.ex. har missförstått kunden, eller om kunden bara har ändrat sin åsikt.

Refaktorisering

Håller koden i konstant bra skick.

2.2 The Planning Game

- Kunden skriver ner features på story cards.
- Utvecklarna delar upp story:na i tasks.
- Utvecklarna uppsattar tidsåtgång för varje taks.
- Kunden väljer tasks för att fylla hela nästa iteration. Resterande tasks sparas till nästa iteration.

2.3 Prioriteringar vid programvaruutveckling

Av de fyra storheterna:

- Bemanning, antal utvecklare
- Tidsåtgång, kalendertid
- Funktioner hos systemet
- Kvalitet på funktionaliteten

Kan man vid programvaruutveckling bara välja tre.
XP håller fast vid tre: bemanning, tidsåtgång och kvalité, men låter funktion variera.
Andra utvecklingsmetoder låter tid variera, vilket kan leda till att det ej är klart i tid.

2.4 Problem vid utvecklingen

I traditionella utvecklingsmetoder är potentiella problem att systemet inte blir klart i tid. Det man gör då är att tillsätta mer personal, men inledningsvis så kommer projektet att stå still och fördröjas ännu mer, då de ”gamla” måste lära upp de nya.

I XP är potentiella problem att vi inte får med alla funktioner. De väsentligaste funktionerna *bör* vara med, då det är dessa man börjar med. Men, om önskvärda funktioner ändå saknas, kan detta lösas genom att tillsätta mer personal. Upplärningen av den nya personalen bör gå snabbare än i traditionell utveckling på grund av parprogrammeringen.

3 Konfigurationshantering

SCM: Software Configuration Management
CVS: Concurrent versions system (CVS) är ett vida spritt och mycket populärt system för versionshantering, speciellt i projekt baserade på öppen källkod. [2]
Git: Git är ett versionshanteringsprogram som skapades 2005 för att hantera källkoden till Linuxkärnan. Linus Torvalds ansåg att inget av de alternativ som fanns att tillgå räckte till vad gäller funktion eller prestanda. [3]

3.1 CVS Overview

Källkoden sparas i ett *repository* på en server. Ändringar görs i lokala *workspace*. Det lokala workspacet skapas via en *check out*, och görs bara en gång. Då kopieras filer från repositoryt till den egna lokala datorn (lokala workspacet).
Efter att utvecklaren har ändrat i någon fil kan hon göra en *commit* för att ladda upp den nya versionen. Utvecklaren kan även göra en *update* för att kopiera de senaste filerna från repositoryt till eget lokalt workspace.

Repositoryt sparar alla versioner av filerna så att man ska kunna gå tillbaka om det behövs. Ett repository består av en eller flera moduler. Vanligtvis så är en modul ett projekt/program.

[2] `http://sv.wikipedia.org/wiki/Concurrent_versions_system`
[3] `http://sv.wikipedia.org/wiki/Git`

3.1.1 Vanliga kommandon

- **checkout**: skapa ett nytt workspace.
- **update**: uppdatera workspacet, dvs. kopiera nya filer från repositoryt till workspacet.
- **kommit**: ladda upp ändrade filer från repositoryt.

Tips

Viktiga arbetsvanor:

- Se till så att den senaste versionen av repositoryt alltid fungerar, dvs. kompilerar och alla test är gröna.
- *Update* ofta, så att man alltid har den senaste versionen.
- *Update* **alltid** innan *commit*.
- *Commit* ofta, så att andra utvecklare kan få senaste versionen.

3.2 Git

Som CVS, men även med ett lokalt repository. Kommandona skiljer sig åt.

3.2.1 Kommandon

- **clone**: ungefär som cvs *checkout*. Skapar både ett lokalt repository samt tillhörande workspace.
- **pull**: ungefär som cvs *update*. Hämtar filer från (globalt) repo till eget workspace.
- **commit**: laddar ändringar till eget (lokalt) repo.
- **push**: ungefär som cvs *commit*. Laddar upp filerna från lokalt repo till globalt repo.

3.2.2 Fördelar med Git och eget lokalt repo

- Du kan arbeta lokalt och commit:a till eget repo utan nätverksåtkomst.
- Du kan skapa egna repositorys lokalt, så att man kan få alla fördelar med version control för eget projekt.
- Om den globala servern krashar, kan man fortfarande jobba på lokalt.

Tips

Precis som att man alltid ska göra en **update** innan **commit** i CVS, så bör man göra **pull** innan man gör **commit** och **push** i Git.

3.3 SCM och XP

Bidrar till

- hantera källkod
- delat kodägandeskap
- simpel integration

- smärtfri refaktorisering
- enkelt att testa
- enkelt att göra releaser
- hantera dokumentation

3.4 Mål med SCM

- Att kunna gå till baka till fungerande ”states”.
- Att ha en överblick över mjukvarans utvecklings-historik.
- Visa vad som beror på vad.
- Hjälpa människor att samordna sitt arbete.

3.5 3 Typiska problem som visar varför SCM behövs

Double maintenance
Problemet är att ha flera identiska kopior av samma mjukvara, då det är svårt och ha alla uppdaterade. Vi löser detta genom att undvika multipla kopior av samma version.

Shared data
Problemet att flera personer beror på och redigerar samma data (kod). När en programmerare gör ändringar i en fil, kan de ändringarna förstöra för andra programmerare.

Simultaneous update
När två personer samtidigt jobbar med samma fil och sedan uppdaterar är det möjligt att en persons uppdateringar blir överskrivna.

4 Testning

4.1 Testnivåer

Hur mycket av systemet är inblandat i testerna?

- Enhetstest, t.ex. en klass
- Modultest, t.ex. ett paket
- Subsystem, t.ex. ett lager
- Systemtest (hela systemet)

Enhetstest och acceptanstest

Man brukar dela upp tester i två kategorier, enhetstest och acceptanstest.

- *Enhetstest* (eller utvecklartest). Testar delar av koden.
- *Acceptanstest* (eller kund test). Testar så att önskade features fungerar och lever upp till kundens önskemål.

4.2 Testdriven utveckling i XP

- Skriv ett test.
- Kompilera testet. Det ska misslyckas eftersom koden som testet anropar inte finns än.
- Implementera bara tillräckligt för att kompilera.
- Kör testet och se det misslyckas.
- Implementera minsta möjliga så att testet blir grönt.
- Kör testet och se så det blir grönt.
- Refaktorisera.
- Om man använder ett SCM: *update* och *commit*.
- Börja om från början!

5 Parprogrammering

Två personer som sitter vid samma maskin. Den som skriver kod brukar kallas för *Föraren* (Driver) och den andra kallas för *Navigatör*.

Föraren

- Skriver kod.
- Beskriver vad han gör, typ ”Här behöver vi en temporär variabel för att beräkna summan."[4]

Navigatören

- Granskar koden efter fel (ex. felstavningar, fel metodanrop, syntaxfel).
- Ger direct feedback på design, metodnamn m.m.
- Håller reda på todo listan.
- Föreslår förbättringar.

5.1 Parprogrammeringstips

- Byt partner ofta.
- Tala i Vi-termer istället för Du-termer. ”Vi gjorde fel här och bör göra såhär istället..”.
- Driver
  - Var lyhörd.
  - Rusa inte iväg, se till så att partnern förstår.
- Navigatören
  - Hitta förarens rytm.
  - Föreslå rollbyte om ni kör fast.

[4] Två fördelar med att beskriva högt vad man gör:

– Vi kan hitta fel/brister i koden bara genom att förklara för någon annan.

– Det håller navigatören engagerad eftersom båda parter kommunicerar.

- Ta Pauser
  Minimum att varje timme: Stå upp, stretcha och titta på något som är mer än en meter bort.
- Var ödmjuk
  - Lyssna och diskutera.
  - Lär av partnern och lär ut till partnern.

6 Enkel design

Kod med enkel design är

- Tydlig och lättbegriplig.
- Har inte någon 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.

Teknik för enkel design

Tydlig lättbegriplig kod är kod man ska kunna första utan att först dechiffrera den.
Detta får vi med t.ex:

- Goda val av namn på t.ex. variabler, klasser, metoder..
- Namnge värden och beräkningar (som variabler och metoder) - istället för att bara koda algoritmen direkt. Alltså att spara delsteg i algoritmen och namnge variabeln så att man enkelt förstår vad som händer.
- 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 och klass skall användas till ett väldefinierat ändamål (så att den är enkel att namnge).
- Namn ska uttrycka vad objektet innehåller, inte hur.

Komplex kod med enkel design

En enkel design kan vara komplex om problemet vi försöker lösa är komplext.
Men designen är ”enkel” om:

- Varje ide är explicit i koden.
- Det inte finns duplicerad kod.
- All komplexitet i koden är motiverad av testfallen.

6.1 Code Smells

Code smells är symptom i koden som antyder om bristfällig design.
Exempel på code smells:

Duplicerad kod
Om samma kod används på flera ställen blir koden bättre om det går att kombinera den duplicerade kod-biten och bara ha den på ett ställe.

Long method
Vi vill inte ha för långa metoder som gör flera saker samtidigt. Bättre att dela upp i flera mindre metoder.
Ett block av kod med en kommentar som berättar vad koden gör kan oftast ersättas av en metod. Metodens namn ska ersätta kommentaren, och då vara tillräckligt bra för förklara i stort sätt samma sak som den tidigare kommentaren.

Large Class
När en klass försöker göra för mycket syns det ofta på för många instans/member variabler. Man bör i så fall försöka bryta ut liknande metoder i en egen klass.

Large Paramer List
Långa parameter-listor är svåra att förstå och underhålla.
Försök att istället låta metoder göra anrop på objektet och på så sätt få tag på värden. Detta är bättre än att skicka in sagda värden som parametrar i metoden.

7 Refaktorisering

Refaktorisering: omstrukturering av koden utan att ändra det yttre beteendet.

Syftet med refaktorisering är att göra mjukvaran enklare att ändra och förstå. Refaktoriseringen ändrar inte det observerbara beteendet av mjukvaran.

Man bör refaktorisera hela tiden när man utvecklar mjukvara. Det är inte något som man avsätter tid till att göra, utan det är en viktig del i själva utvecklingsprocessen som bör ske reflexmässigt.

Syftet med refaktorisering

- Regelbunden refaktorisering håller koden i ”trim”. Utan regelbunden faktorisering så kommer koden att förfalla.
- Refaktoriseringen gör så att koden blir lättare att förstå. Det är bättre att skriva tydlig kod som är lätt att förstå men kanske tar några klockcykler längre tid att köra än att skriva ”snabbare” kod som tar veckor att förstå. Tänk på att i XP äger alla koden så det är mycket möjligt att någon annan kommer att läsa din kod.
- Refaktorisering hjälper dig att hitta buggar. När man refaktoriserar koden så förtydligar man koden och då är det enklare att hitta buggar.
- Refaktoriseringen hjälper dig att programmera snabbare. Utan god design, kan man utveckla mjukvara snabbt tills det att den dåliga designen reducerar tempot. En bra design är avgörande för att bibehålla tempot i utvecklingen.