SIG-Meeting

Context project: Crisis management

2-5

Op 2 mei zijn we met groep 1 en Joost naar SIG geweest. We hebben het hier voornamelijk gehad over refactoring en punten waar we op moeten letten bij het programmeren.Hier volgt een samenvatting van dit gesprek.

1 Stappenplan

1. Hoe moet het werken?

   Wat belangrijk is, is hoe we de code kunnen begrijpen en zorgen dat we een complete test hebben.

   (a) Code begrijpen:

      i. Praten met de maker
      ii. Door de code heen lopen
      iii. zelf een UML gaan maken van hoe de situatie nu is. (Zorg ook dat er een legenda bij zit en classes die heel groot zijn, mogen ook groter afgebeeld worden)
      iv. Dingen als *repast* niet proberen te begrijpen (hier kunnen we toch niks aan veranderen). We hoeven alleen te weten hoe we er gebruik van maken en wat er mogelijk is.
      v. Duidelijke onderverdeling tussen **Server** en **Client** maken. Hoe hangt alles bijv. aan de repast?
      vi. Het plaatje dat we nu hebben laat implement/extend relaties zien. Wat beter en duidelijker is zijn **call relations**

   (b) Tests

      i. Hoe kunnen we van de buitenkant kijken naar het programma? "Soort van schil er om heen"
      ii. Denk aan een black box, het moet automatisch getest kunnen worden. Ga gebruik maken van een testomgeving : *Maven site* in combinatie met *plugin build server* (Jankins)
      iii. vb. scenario opzetten en testen, kijken of resultaten beide keren gelijk zijn.

iv. Het kan handig zijn om gewoon iets te simuleren op bijv. een white board en dan met elkaar te bespreken wat er dan getest moet worden.

v. **Belangrijk!** : Wat ons sterk aangeraden werd is om met metrics te gaan werken. Die kunnen ons veel meer vertellen over de status van de code dan een test van hun. Op die manier kunnen we zien waar grote classes zitten en waar alles samenkomt en in hoeverre onze verbetering ook verstandig zijn. (oude classes moet je bijv. niet altijd gaan aanpassen). De test die zij doen is te breed om voor ons later een duidelijk verschil te kunnen aangeven. Programma dat we hiervoor kunnen gebruiken is: **Sonar**. Dit moeten we zelf installeren en bij elke checkout mee draaien.

2 Tips

- Why refactor?

  Het moet een praktisch nut hebben. Je doet het niet alleen om het mooier te maken, er moet daadwerkelijk iets aan te "verdienen" zijn.

- Naamgeving: Probeer dit zo generiek mogelijk te houden. Maak bijv. gebruik van *frontend/backend*

- Gelijk *unit-tests* schrijven heeft niet veel zin, misschien bedenken we een uur later dat we een bepaalde methode/classen helemaal niet nodig hebben.

- Hoe voorkom je dat groepen elkaar in de weg gaan zitten? Denk aan **waar** ga je iets veranderen en **wat** gaat het worden. Je wil elke dag dat het gentegreerd/getest is

- Features vs. Feasability

  Hier moeten we zelf afwegingen gaan maken in wat het belangrijkste is. Er zal een goede balans in gevonden moeten worden.

- Plugin-architecture:

  Wat zijn de variatie punten, is het run time of niet? Wat zijn plugins en wat zijn gewoon features? Een nieuwe bot/omgeving is over het algemeen een feature. We moet goed kijken of we echt willen gaan werken met een plugin-architecture. Afweging maken in hoeverre iets gewoon configuratie is vs het programmeren. Daarbij een vraag: Wat houd je bijv. in de agent en wat in de environment?

- De test die SIG draait bestaat uit 8 punten genoemd in: Guidance for producers, `http://goo.gl/uw8gHH`

  Deze 8 punten worden op 5 punten gecheckt, is het: analyseerbaar, aanpasbaar, testbaar, modulair en herbruikbaar.

- Let op! Wat vaak bij hun opviel is dat binnen het context project te grote methode worden gemaakt.

3 Slide van hun: Voordat je gaat refactoren?

- Wat is het doel?

  zijn het functionele of technische verbeteringen?

- Hoe voorkom je dat alles stuk gaat?

  zijn er al test? Hoe ga je dit opzetten?

- Hoe ga je het werk binnen je team verdelen?

  wanneer en hoe vaak vind integratie plaats

- Hoe weet je wanneer je het gewenste doel hebt bereikt?

  hoe meet je tijdens refactoring waar je staat?

4 Slide van hun: Pipeline release software

Voor een release doorloopt SIG altijd het volgende. Je kan gelijk zien of alles nog werkt. Hoe verder in het proces hoe langer de stappen ook duren. Op deze manier worden kleine fouten er gelijk aan het begin uitgefilterd en hoef je geen grote algemene test af te breken omdat er ergens een + of - verkeerd staat.

- Compiled + test
- Install
- Dependencies check + unit test
- Database aanmaken
- Voorkant "site" checken
- geautomatiseerd "site" ckecken
- Alle code wordt controleerd of er grote veranderingen zijn.