Refactoring is het op een systematische manier veranderen van de interne structuur van software zonder het observeerbare gedrag van de software te veranderen. Refactoring helpt de meest recente inzichten over het ontwerp van het systeem weer te geven in de code en gaat verval van de structuur tegen. Een introductie.

achtergrond

Ontwerpverbetering in bestaande code

Refactoring

Bij het bedenken van een ontwerp voor een systeem is het belangrijk om diverse verantwoordelijkheden eenduidig over de onderdelen van een systeem te verdelen. Echter, vanaf de implementatie begint de structuur van het systeem te vervallen: de verantwoordelijkheden in de ontwerpdocumentatie zijn toch niet zo duidelijk als ze lijken, details zijn over het hoofd gezien en om die ene bibliotheek of dat ene raamwerk te kunnen gebruiken moet een deel van het ontwerp er anders uitzien. Daarnaast zal ook de ontwerpdocumentatie vaak niet bijgehouden worden. Als een systeem uiteindelijk bij oplevering een goede structuur heeft, zal de structuur veranderen tijdens onderhoud. Het is niet de bedoeling dat onderhoud de werking van het systeem aantast. In de praktijk wordt er daarom vaak code om de originele software heen gemaakt. Bij elke nieuwe verandering wordt de structuur zwakker en moeilijker te overzien, zodat elke volgende wijziging duurder wordt.

Een probleem met onderhoud is, dat het repareren van een defect nieuwe defecten introduceert. Volgens [Brooks] ligt de kans, dat reparatie van een defect een nieuw defect oplevert, tussen de 20 en 50 procent. Als een defect is gelokaliseerd, wordt het defect ook lokaal gerepareerd, terwijl de consequenties van de reparatie vaak betrekking hebben op een veel groter deel van het systeem. Degene die de reparatie uitvoert is vaak niet degene die de code heeft gemaakt, en zal dus minder op de hoogte zijn van de complete betekenis van de defecte code en zijn omgeving.

Voordat we iets kunnen gaan doen aan slechte code zullen we in staat moeten zijn om slechte code te herkennen. Slechte code is code die het lastig maakt om een programma te begrijpen, te repareren en uit te breiden. Het is handig om een aantal kenmerken te hebben waaraan men deze code kan herkennen. Deze kenmerken hoeven niet volledig te zijn in de zin dat we er alle slechte code mee moeten kunnen herkennen. De kenmerken die we kunnen gebruiken noemen we ‘code smells’. Code smells geven aan dat de code ‘stinkt’. ‘Gedupliceerde code’ is een voorbeeld van zo’n code smell. Gedupliceerde code ontstaat vaak door gebruik van copy en paste om stukken programma opnieuw te gebruiken. Het ontstaat vaak ook bij het repareren van defecten; als een defect gelokaliseerd is, wordt de operatie waar het defect in zit gekopieerd naar een nieuwe operatie en wordt de bug in de nieuwe operatie gerepareerd. Als er in de gedupliceerde code een defect verborgen zit, zal deze meestal niet bij alle gedupliceerde stukken tegelijk optreden. Het stuk waar het defect optreedt wordt gerepareerd, terwijl een aantal kopieën van hetzelfde defect in de overige stukken verborgen blijft. Gedupliceerde code zorgt dus voor programma’s die moeilijk te begrijpen, onderhouden en te repareren zijn.

Aanpassingen aan een systeem hebben, zoals gezegd, het risico in zich om nieuwe defecten te introduceren. Daarom moeten wijzigingen op een systematische manier worden doorgevoerd. Refactoring is zo een systematische aanpak. Vóór wijzigingen aan het gedrag van een systeem wordt de structuur aangepast, zodat de beoogde wijziging makkelijker door te voeren is.

WISKUNDIGE EXPRESSIES

De activiteit refactoring bestaat uit het uitvoeren van een aantal één of meer refactorings. Een refactoring is een transformatie van broncode naar broncode. Een refactoring is vergelijkbaar met het veranderen van de factoren in de volgende wiskundige expressie: $(x+1)*(x+2)$ kan ook worden geschreven als $x^2 + 2x + x + 2$ en dat kan weer worden geschreven als $x^2 + 3x + 2$. De uitkomst van beide expressies als er een waarde voor x wordt gegeven is dezelfde, maar de structuur van de laatste expressie is vele malen eenvoudiger. Zo is het ook met refactoring. Een stuk programmacode verandert niet van observeerbaar gedrag, maar wel van structuur. Het nut van refactoring is dan ook gelijk aan dat van het vereenvoudigen van wiskundige expressies – het maakt veranderingen eenvoudiger.

Refactoring beschermt een investering in werkende

code, terwijl de code klaar wordt gemaakt of gehouden voor de nabije toekomst. Een veel gehoord bezwaar tegen het aanpassen van bestaande code is: “ja, maar het programma werkt toch?”. Mijn reactie is dan dat het programma nu wel werkt, maar werkt het morgen nog? Het programma werkt, maar de structuur is kapot. Een bijkomend voordeel van refactoring is, dat het makkelijker is om moeilijke beslissingen uit te stellen. Het is bijvoorbeeld vaak moeilijk om een goede naam voor elementen in een ontwerp te vinden, bijvoorbeeld de naam van een operatie, een parameter of een module. In plaats van te blijven aarzelen, kan een voorlopige naam worden gekozen. Deze kan later altijd nog door refactoring worden veranderd.

ENCAPSULEER ATTRIBUUT Om duidelijk te maken waar het om gaat geven we een beschrijving van een eenvoudige refactoring, met daarbij de toepassing op een voorbeeldprogramma in de object-georiënteerde taal Java. Deze refactoring is beschreven in [Fowler]. Een kleine refactoring is ‘encapsuleer attribuut’. Een van de belangrijkste principes van object-georiënteerd programmeren is ‘encapsulatie’. Dit betekent dat data in een klasse nooit zo maar voor de buitenwereld toegankelijk gemaakt wordt. De manier om dit te doen is de toegang tot attributen in een object alleen via berichten toe te staan. Er zijn echter object-georiënteerde talen, zoals Java en C++, die toestaan dat attributen niet ingekapseld zijn. Deze talen staan het toe om attributen ‘public’ te maken, zodat andere klassen in het systeem de data in een klasse zo maar kunnen veranderen. Hierdoor kan deze klasse bijvoorbeeld in een ongeldige toestand terechtkomen, doordat er geen controle kan worden uitgevoerd op de juistheid van de waarde die is toegekend aan het attribuut. De oplossing voor dit probleem is het implementeren van public ‘methods’ die een bericht voor het schrijven en een bericht voor het lezen van een attribuut implementeren. Het attribuut kan dan voor de buitenwereld verborgen worden door het attribuut ‘private’ te maken. De stappen om de refactoring ‘encapsuleer attribuut’ systematisch uit te voeren zijn als volgt:

- Maak lees- en een schrijf-methoden voor het attribuut.
- Zoek alle code in andere klassen die refereren aan het attribuut. Als de code het attribuut leest, vervang de code door een aanroep van de zojuist gedefinieerde leesmethode. Als de code een waarde toekent aan het attribuut, vervang de toekenning dan door een aanroep van de schrijfmethode.
- Compileer en test na elke verandering.
- Als alle code die het attribuut gebruikt is aangepast, definieer het attribuut als ‘private’.
- Compileer en test.

Als voorbeeld bij encapsuleer attribuut introduceren we de klasse Persoon. De klasse persoon heeft een attribuut naam van het type String. Dit attribuut representeert de voor- en achternaam van een persoon. De klasse Persoon ziet er aan het begin van de refactoring zo uit:

```
class Persoon {
        public String naam;
}
```

Uit een verzoek tot verandering van de applicatie waarin Persoon wordt gebruikt, blijkt dat er expliciet onderscheid gemaakt moet gaan worden tussen de voornaam en de achternaam van de persoon. Niet alle cliënt code heeft echter belang bij het onderscheid tussen voor- en achternaam. Het systeem moet dus zodanig aangepast worden, dat voornaam en achternaam kunnen worden gebruikt, terwijl er ook gebruik moet kunnen worden gemaakt van de volledige naam. Om dit mogelijk te maken, wordt de refactoring encapsuleer attribuut toegepast, zodat naam een lees- en schrijfmethode krijgt. De leesmethode noemen we getNaam en de schrijfmethode setNaam. Methode getNaam heeft geen parameters en geeft de naam als string terug. Nu vervangen we in cliënten van Persoon de code die refereert aan de string naam door aanroepen van getNaam of setNaam. Bij het vinden van gebruikers van Persoon.naam kunnen we slim gebruik maken van de compiler. Door naam alvast private te maken zal de compiler melding maken van elke cliënt die probeert een waarde aan naam toe te kennen, of een waarde uit naam te lezen.

```
class Persoon {
  private String naam;
  public String getNaam() // lees methode
{return naam;}
  public String setNaam(String eenNaam) //schrijf methode
{naam = naam;}
}
```

Na deze refactoring kan de volgende stap genomen worden, het stapsgewijs omzetten van naam in voornaam en achternaam. De cliënten van Persoon die alleen getNaam gebruiken hoeven niet te worden aangepast, alleen de implementatie van getNaam wordt veranderd in:

```
{ return (voornaam + “ “ + achternaam) }
```

De cliënten die setNaam gebruiken moeten wel worden aangepast, omdat ze in plaats van setNaam gebruik moeten maken van de nieuwe methoden setVoornaam en setAchternaam. Ook hier kan de compiler worden ingezet om de betreffende cliënten te vinden, net als we bij het zoeken van gebruikers van Persoon.naam hebben gedaan. Persoon.setNaam() kan tijdelijk als private gedeclareerd worden, en de compiler komt dan netjes met de plaatsen waar deze gebruikt wordt.

Door het uitvoeren van een refactoring vóór de introductie van voornaam en achternaam was deze wijziging

eenvoudiger uit te voeren; alleen de cliëntcode die een waarde toekende aan Persoon.naam moest functioneel worden aangepast.

LINT Er zijn een aantal min of meer standaard gereedschappen die een ontwikkelaar kunnen helpen bij het refactoren. Er zijn ook gereedschappen die speciaal zijn gebouwd voor het uitvoeren van refactoring.

Bij handmatig refactoren is het van belang dat gemaakte syntaxfouten snel gecorrigeerd kunnen worden. Een snelle parser voor de betreffende taal is heel belangrijk: als je een dag moet wachten om te zien of een refactoring goed is uitgevoerd is het niet erg aanlokkelijk om veranderingen aan te brengen. Cross-reference-mogelijkheden in de ontwikkelomgeving zijn zeer nuttig, vooral als ze rekening houden met de syntax van de code. Het is bijvoorbeeld handig om de ontwikkelomgeving te kunnen vragen: "welke andere methoden gebruiken deze methode?".

Een voorbeeld van een ondersteuningstool is Lint. Lint kan defecten in de code, vergissingen en overbodige code vinden. Lint en varianten van lint zijn er onder andere voor C, C++ en Smalltalk. IBM VisualAge (voor Java, C++ en Smalltalk) biedt de mogelijkheid om interactief naar referenties van bijvoorbeeld een class of methode te vragen.

Een tool of 'zoek en vervang'?

Wat maakt een refactoring tool beter dan zoeken en vervangen van tekst? Stel we hebben een class A met de methode test en een class B ook met een methode test. Class A heeft de volgende definitie:

```
class A {
  public void test(b B)
{ b.test(); }
}
```

Als we in de class-definitie van A het woord test zouden zoeken en vervangen, dan wordt A:

```
class A {
  public void testen(b B)
{ b.testen(); } // aanroep van b.test() gebroken
}
```

Waardoor de aanroep van b.test() in voorheen A.test() gebroken is. Een refactoring tool had rekening kunnen houden met het feit, dat b.test() en A.test() twee compleet verschillende dingen zijn, en had dus slechts A.test een andere naam gegeven. Een zoek en vervang operatie kan dat niet.

AUTOMATISCH REFACTORING-GEREEDSCHAP Een goede refactoring tool voldoet aan de volgende criteria:

- integratie in de ontwikkelomgeving;
- snelheid;
- correctheid;
- interactiviteit (ontwerpers hebben voldoende invloed op hoe de software eruit gaat zien).

Als aan bovenstaande criteria voldaan is, wordt refactoring met gereedschap voor de ontwikkelaar een stuk aantrekkelijker dan refactoring zonder gereedschap. De hoeveelheid met de hand aan te passen code wordt minder en door de correctheid ingebouwd in het gereedschap is er minder noodzaak voor testen. Daarnaast verhoogt gereedschap de productiviteit. Als refactorings het gedrag niet kunnen veranderen zijn ze veilig, en hoeft er dus minder getest te worden. Bovendien maakt een refactoring gereedschap het mogelijk om 'what if'-analyses uit te voeren op de code. Zeker als dit gereedschap een 'undo'-faciliteit heeft, kunnen zeer eenvoudig alternatieve indelingen van de software worden bekeken. Als het niet bevalt is een paar keer op undo klikken voldoende om weer bij de oude toestand uit te komen. Een werkend refactoring gereedschap met deze eigenschappen is de Refactoring Browser voor (VisualAge en Visualworks) smalltalk. Deze gratis tool is te vinden op http://st-www.cs.uiuc.edu/~brant/Refactory/RefactoringBrowser.html.

Met automatische refactoring verdwijnt niet de menselijke creativiteit. Het vereenvoudigt alleen een aantal tijdrovende routinetaken die noodzakelijk zijn bij het verbouwen van een systeem. De programmeurs/ontwerpers bepalen zelf welke refactoring waarop wordt toegepast. Door de ondersteuning kunnen zij zich meer bezighouden met aanpassingen op ontwerpniveau.

STRUCTUUR EN CONTROLE Refactoring is niet beperkt tot één soort programmeertaal (bijvoorbeeld objectgeoriënteerde programmeertalen). In elke programmeertaal kan een programma in stappen worden herschreven. Ook voor modellen en diagrammen kunnen dit soort stappen worden gedaan. Een systeem waarop refactoring regelmatig wordt toegepast is eenvoudiger te onderhouden, omdat de software de meest recente inzichten over het systeem reflecteert, in plaats van de inzichten bij het ontwerp of de eerste oplevering. Refactoring is natuurlijk geen excuus om te gaan hacken. Het is belangrijk dat refactoring gestructureerd en gecontroleerd plaatsvindt.

Willem van den Ende
is onderzoeker bij het Software Engineering Research Centre en bereikbaar via ende@serc.nl

Literatuur

[Brooks] The Mythical man-month – essays on software engineering, Frederick P. Brooks jr., 20th anniversary edition, Addison Wesley, 1995

[Fowler] Refactoring - improving the design of existing code, Martin Fowler, Addison Wesley Longmann, 1999