STUDI DAN PENGEMBANGAN METODE REKAYASA PERANGKAT LUNAK BERORIENTASI ASPEK

Lola, ST, MT
[1]Program Studi Teknik Informatika, FTI, Institut Teknologi Budi Utomo Jakarta,
lola.rezak@gmail.com

Abstract
This paper researching the conception of aspect-oriented software engineering, including requirements engineering, analysis and design phase. Investigation focused mainly on handling separation of concerns, crosscutting concerns, functional and non-functional concerns identification, specification, and its modeling notation.
The research started by observing some of the proposed activities for requirements engineering, analysis and design literature, and the concept of aspect-oriented paradigm. Most of the literatures were found from web-site, such as working paper for lecturing, workshop or conference.
This paper proposes a new aspect-oriented software engineering method, focus on (1)requirement engineering, (2)analysis and design phase. In order to define this new method, some of the previous methods are explored and analyzed. The characteristic of this new method will be analyze and measured using soft measurement. Furthermore, the method will be used on simple case study to show the utilizing of this method. Aspect-oriented software engineering will be done easier by using this new method than previous method, for small size to medium size system development at common problem domain. Supposing the specification of concerns and module become more modular and have a good usability that bring the software development and evolution easier and faster by using aspect-oriented paradigm.
The result of this paper is a new aspect-oriented software engineering method for requirement engineering, analysis and design phase, and a simple case study simulation application.
***Keyword**: aspect-oriented, crosscutting concerns, requirement engineering, analysis and design.*

1. PENDAHULUAN

Pemrograman Berorientasi Objek (PBO) telah menjadi suatu paradigma pemrograman yang sangat baik dalam memodelkan permasalahan dunia nyata ke dalam abstraksi objek-objek yang mengandung sifat dan data. Walaupun demikian, PBO masih memiliki kelemahan dalam melakukan pemisahan atau lokalisasi *croscutting concerns* yaitu *concerns* yang memiliki kepentingan yang tumpang tindih (berpotongan) terhadap *concerns* lain. Sebagai konsekuensinya, pembangunan dan evolusi perangkat lunak yang dibangun dengan paradigma PBO masih mengalami kendala atau kesulitan, disebabkan teknik pemisahan *concerns* yang kurang dan tingkat modularitas yang rendah.

Ketidakmampuan untuk lokalisasi *concerns* dan adanya *crosscutting concerns* (concerns yang saling berpotongan, biasanya berupa *concerns* yang sifatnya non-fungsional) ditandai dengan adanya ***Scattering*** dan ***Tangling***. *Scattering* adalah kondisi dimana ada kode yang mirip terdapat pada banyak bagian di aplikasi. *Tangling* adalah kondisi dimana dua atau lebih concern diimplementasikan dalam sebuah modul yang sama (21). Implikasi dari kedua kondisi diatas adalah:

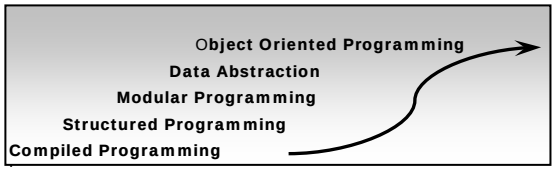
1. Kesulitan penelusuran.
2. Tingkat produktivitas rendah
3. Tingkat guna ulang modul rendah
4. Kualitas kode rendah
5. Sulit melakukan perubahan pada sistem.

Dalam PBO, kondisi *scattering* dan *tangling* sering terjadi sebagai akibat dari kelemahannya menangani *crosscutting concerns*

(The tyranny of the dominant decomposition[1]). Pemrograman Berorientasi Aspek (PBA) adalah sebuah teknologi baru yang mampu melakukan lokalisasi *crosscutting concerns* ke dalam satu unit yang disebut ***aspect*** [2]. *Aspect* adalah sebuah unit modular untuk implementasi permasalahan *crosscutting concern*.

Seiring dengan perkembangan *tools* PBA, konferensi yang membahas metodologi Rekayasa Perangkat Lunak Berorientasi Aspek (RPL-BA) telah memasuki tahun ke-4 pada Maret 2005. Metode tersebut cukup bervariasi baik dari pendekatannya maupun dari domain kasusnya.

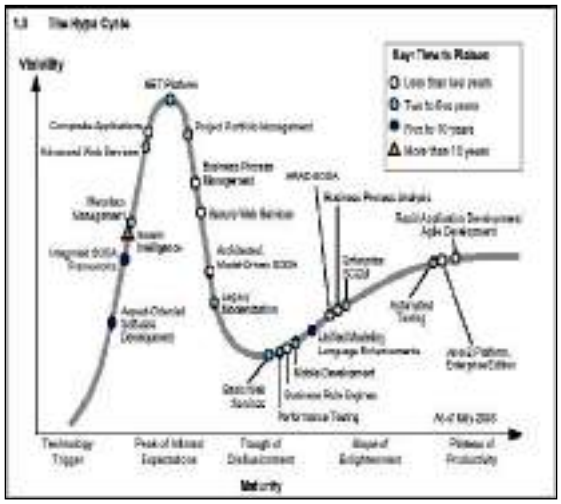
IBM *Software Group* dalam presentasinya pada konferensi internasional untuk *Aspect Oriented Software Development* (AOSD) ke-3 menyatakan *"AOSD is the next step in the series of advances in Software Engineering Modularity "* (30) yang diilustrasikan seperti gambar 1.



Gambar 1. Ilustrasi Perkembangan Modularitas Rekayasa Perangkat Lunak (30)

Penelitian oleh Gartner Research pada tahun 2003 (30) tentang perkembangan teknologi pembangunan aplikasi yang dilihat dari dua dimensi *(visibility* dan *maturity*), menunjukkan bahwa AOSD masih berada pada tahap awal sebuah teknologi baru (lihat gambar 2).

Visibility menunjukkan tingkat popularitas teknologi tersebut. Semakin tinggi tingkat *visibility*, semakin dikenal teknologi tersebut. *Maturity* menunjukkan tingkat kematangan teknologi tersebut.



Gambar 2 *Gartner Hype Cycle For Application Development 2003* (30)

Hasil penelitian tersebut sejalan dengan hasil penelitian dari SEI (*Software Engineering Institute*) *Independent Research and Development Projects and Report on Emerging Technologies and Technology Trends* (6). Menurut hasil penelitian SEI, AOSD merupakan salah satu dari 13 teknologi yang akan popular di masa akan datang. Bahkan IBM telah menaruh perhatian besar terhadap teknologi ini, sehingga menjanjikan akselerasi kemajuan teknologi ini.

2. METODOLOGI

2.1 PEMROGRAMAN BERORIENTASI ASPEK (PBA)

Penelitian-penelitian yang berkaitan dengan ide-ide PBA telah dilakukan sebelum nama PBA itu sendiri diperkenalkan oleh Gregor Kiczales (22). *Adaptive Programming* (AP)[3] yang dikenalkan pada tahun 1991 merupakan instans awal dari PBA.

PBA dibuat untuk menangani permasalahan yang sering dihadapi dalam pemrograman, yaitu kondisi *scattering* dan *tangling*. Kedua kondisi tersebut dapat dilihat

[1] *Concerns* tidak dominan dihilangkan dan dikomposisi dalam *concerns* yang dominan pada kondisi *crosscutting concerns.*

[2] Untuk selanjutnya, *aspect* dalam bahasa inggris menyatakan unit implementasi aspek. Aspek dalam bahasa indonesia mengandung pengertian sebuah *crosscutting concerns* berupa kebutuhan (*requirements*).

[3] Program didekomposisi dalam beberapa *crosscutting building blocks* yang terdiri dari representasi struktur objek ditambahkan *structure-shy behaviour*.

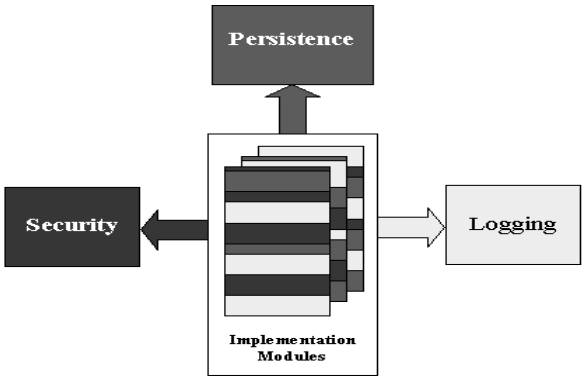
sebagai akibat dari tidak mampunya teknik yang digunakan untuk mengelompokkan suatu *concern* tertentu menjadi sebuah unit yang modular. Teknik pemrograman yang ada diyakini telah mampu melakukan modularisasi *concerns* ke dalam unit-unit yang modular, yang biasanya masih merupakan kebutuhan fungsional. Tetapi modularisasi *crosscutting concerns* masih sulit dilakukan. *Crosscutting concerns* biasanya merupakan kebutuhan yang sifatnya non-fungsional (NF).

Kebutuhan NF biasanya mempunyai kepentingan yang tersebar maupun berpotongan pada unit-unit fungsional. Kondisi ini sering disebut sebagai *crosscut concerns* (*crosscutting concerns*). PBA diharapkan dapat melakukan modularisasi kondisi *crosscut concerns* dengan lebih baik. Pada kasus tertentu, rancang ulang terhadap suatu sistem dapat membuat *crosscuting concern* menjadi suatu unit kelas/ prosedur/ fungsi (*core units*[4]).

PBA merupakan ekstensi dari bahasa pemrograman yang telah ada, artinya konsep PBA tidak terbatas pada bahasa pemrograman dan teknik pemrograman tertentu saja. Dalam implementasinya, PBA berkolaborasi dengan bahasa pemrograman yang didukungnya, sehingga sebuah program yang dibuat dengan paradigma PBA akan terdiri dari modul aspektual dan fungsional/ *core units*.

2.2 *Separation of Concerns (SOC)*

Sebuah sistem dapat dipandang sebagai kumpulan dari *concern* (lihat gambar 3). *Concerns* dapat diartikan sebagai bagian dari tujuan, konsep atau *area of interest* (21).



Gambar 3. Modul Dipandang Sebagai Kumpulan *Concern.*

Manusia tidak mampu untuk memusatkan perhatiannya pada beberapa *concern* dalam suatu waktu. Untuk suatu sistem yang kompleks perlu dilakukan pemisahan *concerns* agar memudahkan rekayasa perangkat lunak. pemisahan *concerns* biasa disebut *separation of concerns* (SOC).

SOC dapat diterapkan pada berbagai bidang dan tingkatan. Untuk rekayasa perangkat lunak, SOC dapat di kategorikan dalam 2 (dua) golongan (24):

1. Proses, yaitu: memisah-misahkan *concerns* berdasarkan aktivitas dan tanggung-jawabnya. Contoh: kualitas perangkat lunak, fase-fase dalam *waterfall*, dan lain-lain.
2. Produk, yaitu: memisah-misahkan *concerns* berdasarkan kebutuhan produk yang dihasilkan. Contoh: fungsionalitas, unjuk kerja, antarmuka, dan lain-lain. Pembahasan SOC selanjutnya lebih ditekankan pada golongan produk.

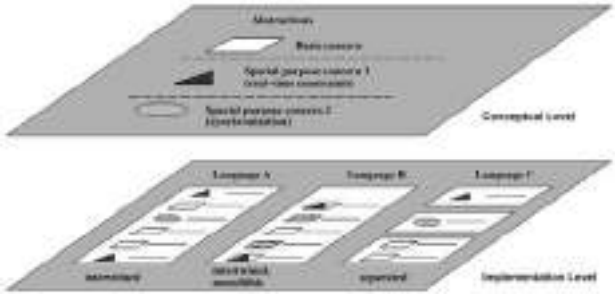
Ada 2 (dua) tingkatan (*level*) SOC (15). Kedua tingkatan ini termasuk dalam SOC golongan produk (lihat gambar 4). Kedua tingkatan tersebut adalah:

1. *Conceptual level*. *Concern* pada tingkatan ini sebaiknya menyediakan definisi dan identifikasi konsep yang jelas untuk tiap *concern*, yang membedakan dengan *concerns* lainnya. Sehingga definisi dan identifikasi tersebut menjamin setiap *concern* tersebut

[4] Untuk selanjutnya, *core units* menyatakan unit implementasi dari kelas/ prosedur/ fungsi, dengan pengertian bukan kebutuhan aspektual.

adalah primitif, bukan merupakan komposisi dari beberapa *concern*.

2. *Implementation level*. SOC perlu menyediakan suatu cara organisasi yang dapat melakukan isolasi *concern*. Tujuan akhir tingkatan ini adalah memisah-misahkan blok-blok kode yang memiliki *concern* yang berbeda, dan menyediakan tingkat kopling yang rendah antar blok tersebut.



Gambar 4. *Separation of Concern* Pada Dua Tingkatan Yang Berbeda (15)

Concerns yang telah diidentifikasi pada *conceptual level*, biasanya dipetakan pada *implementation level* melalui bahasa pemrograman. *Concerns* pada *conceptual level* yang tidak primitif, dapat memberikan efek pada *implementation level*, sehingga implementasi mencoba untuk melibatkan/ mengatur beberapa *oncern* yang berbeda sekaligus. SOC pada *conceptual level* berfungsi terutama untuk mengatur kompleksitas RPL.

SOC pada *implementation level* masih jarang dilakukan, karena bahasa pemrograman yang mendukung masih sedikit. Akibatnya, kode-kode implementasi menjadi *monolithic* dan/atau *intertwined*.

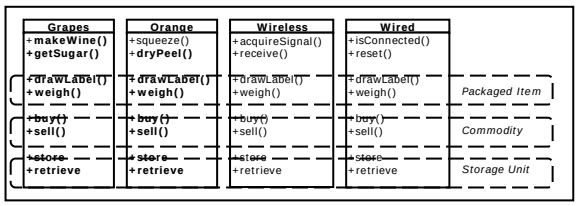
Secara singkat, keuntungan yang diperoleh dengan menerapkan SOC pada kedua tingkat tersebut adalah (15):

1. Menyederhanakan pemrograman.
2. Memudahkan pengertian *concern*.
3. Menghasilkan tingkat kopling yang rendah sehingga meningkatkan fleksibilitas dan guna ulang.
4. Memecahkan masalah *inheritance anomalies*

2.3 *Crosscutting Concerns*

Bahasa pemrograman yang ada saat ini telah mampu melakukan SOC dengan baik dengan melakukan dekomposisi ke dalam unit-unit fungsional. Tetapi ada juga *concerns* yang sulit dilokalisasi / diisolasi secara modular. Biasanya *concerns* seperti ini memiliki kepentingan terhadap beberapa *concerns* lainnya, disebut *crosscutting concern*.

Crosscutting concern dapat diartikan sebagai sebuah *concern* yang memiliki kepentingan terhadap beberapa *concerns* lainnya, yang dapat menghasilkan organisasi kode program yang *intertwined*, atau yang sering disebut kondisi *scattering* atau *tangling*. Sebagai contoh, *concern* produk dapat terdiri dari produk buah-buah dan alat-alat elektronik. *Concern* tersebut dapat dimodularisasi menjadi kelas buah dan kelas elektronik. Tetapi dari sudut pandang pengusaha, *concerns* penyimpanan ataupun transaksi untuk tiap produk adalah sama, tidak secara khusus tergantung pada produk tertentu, sehingga *concerns* tersebut memiliki kepentingan terhadap beberapa *concerns* produk sekaligus (lihat gambar 5).



Gambar 5 Contoh *Crosscutting Concerns* (18)

Pada paradigma PBO, permasalahan di atas dapat diatasi dengan melakukan *implementing* atau *inheriting* fungsi-fungsi yang terkait. Permasalahan yang dihadapi dengan melakukan hal tersebut adalah semua kebutuhan atau *concerns* harus dapat diidentifikasikan sejak awal kegiatan rekayasa perangkat lunak (18).

Untuk lebih jelas menunjukkan kondisi kode *intertwined*, berikut ini adalah contoh kode program yang melibatkan *concern synchronization* dan *queue* (15). Kode program

sinkronisasi masuk dalam tubuh utama algoritma *insert queue*.

```
void BoundedQueue::Insert(Element *el){

// Synchronization
pthread_mutex_lock (&qlock_insert);
while (nelements == MAX) {
cout << pthread_self() << " Wait on
Insert";
pthread_cond_wait (&q_notfull,
&qlock_insert);
// Synchronization
pthread_cond_signal (&q_notempty);
pthread_mutex_unlock (&qlock_insert);
// End of synchronization
}
nelements++;
```

Walaupun kode program diorganisasikan secara teratur, kondisi kode program seperti diatas dapat menyebabkan masalah sebagai berikut (15):

1. Kode program menjadi kompleks.
2. Kode program sulit dimengerti.
3. Pemeliharaan dan modifikasi sulit, sebab tingkat kopling sangat tinggi.
4. Khusus untuk PBO, dapat menyebabkan *inheritance anomalies*.

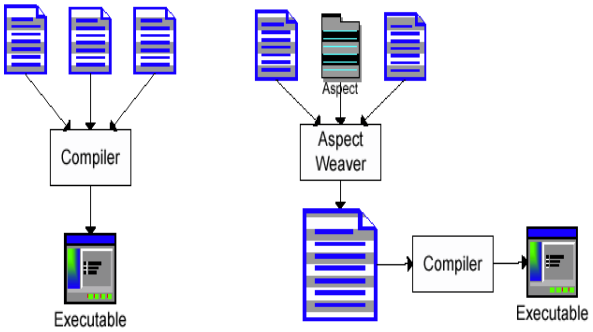
Terdapat 2 (dua) jenis *crosscutting concerns* (18):

1. *Static crosscutting technique*. Bila *crosscutting concerns* tersebut berperilaku/ memberikan efek yang sama terhadap semua *concerns* yang terkait. Contohnya adalah gambar 5.
2. *Dynamic crosscutting technique*. Bila *crosscutting concern* tersebut berperilaku/ memberikan efek tergantung dari *concerns* yang akan dipengaruhi. Contoh, *concern* pembayaran dapat berbeda tergantung pada produk tertentu.

PBA merupakan paradigma pemrograman yang dapat melakukan modularisasi *croscutting concerns* ke dalam modul *aspect*.

2.4 *Weaving*

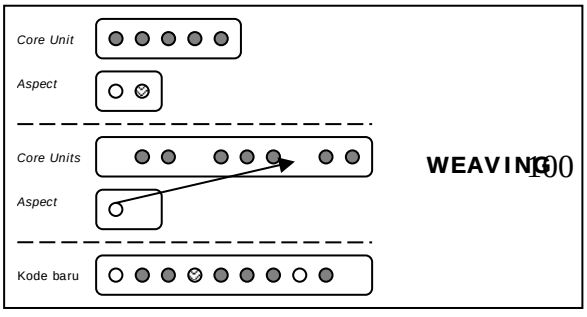
Perbedaan PBA dengan paradigma pemrograman tradisional adalah pada PBA dilakukan *weaving* terhadap *core units* dan *aspects* sehingga menghasilkan sebuah kode baru, kemudian dilakukan kompilasi yang menghasilkan sebuah program yang dapat dieksekusi (lihat gambar 6).

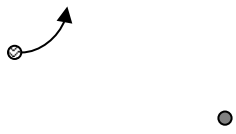


Gambar 6. (Kiri) Paradigma Pemrograman Tradisional , (Kanan) Paradigma PBA Untuk *Static Weaving* (21)

Weaving adalah suatu proses menggabungkan unit *aspects* dan *core units*. Dengan melakukan *weaving*, unit *aspect* dapat mempengaruhi *behaviours* sekumpulan *core units*. Implementasi konsep PBA pada bahasa pemrograman tertentu memiliki algoritma/ kemampuan *weaving* yang berbeda-beda.

Ilustrasi *weaving* dapat dilihat pada gambar 7. Gambar bulatan-bulatan kecil pada *core unit* adalah mewakili urutan instruksi. Gambar bulatan-bulatan kecil pada unit *aspect* adalah potongan-potongan instruksi yang merupakan *concern* yang mempengaruhi *core unit*. Sebuah potongan instruksi pada unit *aspect* dapat mempengaruhi beberapa tempat sekaligus pada *core unit*. *Aspect* digunakan untuk melakukan lokalisasi/ modularisasi *crosscutting concerns*. Untuk melakukan perubahan aspektual, cukup mengubah satu bagian pada unit *aspect* saja. Jika dibandingkan dengan paradigma pemrograman tanpa *aspect*, maka setiap bagian dari *core unit* harus dilakukan perubahan. Setelah proses *weaving* dilakukan, maka menghasilkan kode baru yang telah disisipkan *concern aspect*.





Gambar 7. Ilustrasi *Weaving* Melakukan Penyisipan Kode-Kode Baru Pada *Core Unit*

Berikut ini adalah contoh kasus sederhana dalam bentuk kode program (7) untuk melihat hasil proses *weaving* dengan bahasa pemrograman AspectJ. Kolom sebelah kiri adalah kondisi awal, kolom sebelah kanan adalah kondisi setelah dilakukan *weaving* (disederhanakan) terhadap *aspect* dan *core units.*

```
Package kasus1;
public class A {
  public int IntA,
IntB;
  public String str;
  int a(int x) {

  system.out.println
("A.a");
    return b(x);
  }
  int b(int x) {

  system.out.println
("A.b");
    return x;
  }
}
public aspect
showcase {
  pointcut
int_A_a_int():
    call (int
A.a(int));

  before():
int_A_a_int() {

  system.out.println
("Before");
  }
}
public static void
main (String args[])
{
  D = new A();
  IntA = D.b(8);
  T = new A();
  IntB = T.a(5);
}
```

```
package kasus1;
public class A {
  public int IntA,
IntB;
  public String str;

  int a(int x) {

  system.out.println
("A.a");
    return b(x);
  }

  int b(int x) {

  system.out.println
("A.b");
    return x;
  }
}

public static void
main (String args[])
{
  D = new A();
  IntA = D.b(8);
  T = new A();
  system.out.println
("Before");
  IntB = T.a(5);
}
```

Reserved words pada contoh program di atas sangat spesifik untuk bahasa pemrograman tertentu, sehingga tidak akan dijelaskan secara detil di sini. Pada program di atas terdapat 3 (tiga) buah unit, yaitu: unit kelas A, unit *aspect* showcase, dan unit utama. Pada unit *aspect*, dideklarasikan sebuah pointcut int_A_a_int() berupa pemanggilan dari method int A.a(int). Kemudian didefinisikan *behavior poincut* tersebut adalah before(): int_A_a_int() yang menunjukkan bahwa sebelum dilakukan method int A.a(int), instruksi system.out.println("Before") harus dijalankan terlebih dahulu.

Hasil dari proses *weaving*, dapat dilihat pada sisi kanan kolom contoh program. Objek D dan T merupakan instan dari kelas A. Sebelum objek T menjalankan method T.a(5), disisipkan perintah system.out.println("Before") yang berasal dari *behaviour* unit *aspect.*

Contoh kode program tersebut di atas memang masih sangat sederhana untuk menunjukkan kekuatan dari paradigma BA dalam menurunkan tingkat kopling. Walau demikian, contoh sederhana tersebut di atas dapat menunjukkan bahwa dengan adanya *aspect*, *concern* kelas A dengan *concern* showcase memiliki tingkat kopling yang rendah. Tingkat kopling yang rendah bila menggunakan paradigma BA tersebut semakin terlihat bilamana hubungan antar *concern* tersebut semakin komplek.

Ada dua jenis proses *weaving*:

1. *Static Weaving*: *weaving* yang dilakukan sebelum *run-time*. Menggabungkan kode *aspects* dengan *core units* menjadi kode baru, sehingga tidak dapat lagi dikenali kode *aspects* dan *core units*-nya. (lihat gambar 6).
2. *Dynamic Weaving*: *weaving* yang dilakukan saat *run-time* jika dibutuhkan, sehingga kode program *aspects* masih tetap dipertahankan keberadaannya dari *core units*.

2.5 *Aspect*

Aspect adalah unit untuk membungkus *behaviours* yang dapat mempengaruhi atau memberikan efek terhadap sekumpulan *core units*. *Aspects* adalah unit modular hasil lokalisasi *crosscutting concerns.*

Menurut Gregor Kiczales (26):

"*aspects are analogous to cross-organizational teams within an organization that crosscuts an organization's traditional hierarchical structure*".

Kalimat di atas dapat dijelaskan sebagai berikut. Seorang manager perusahaan

memerintahkan ‘Setiap orang yang terkait pada produk X harus memperhatikan kualitas dengan baik hingga konsumen Y puas’. Kalimat ‘Setiap orang yang terkait pada produk X’ dapat disebut sebagai *pointcut* sebab ia mendefinisikan orang yang ada pada organisasi (*join point*). Kalimat ‘harus memperhatikan’ dapat disebut sebagai *advice* sebab ia mengubah sikap dari setiap orang di organisasi.

Agar dapat dengan mudah membedakan antara unit *aspect* dan *core units*, Kiczales membuat klasifikasi sebagai berikut (18):

1. *Component*, dibuat jika dapat dienkapsulasi dengan sempurna, sebagai sebuah unit fungsional sistem. Dienkapsulasi dengan sempurna artinya adalah dapat dikelompokkan dengan baik, mudah diakses dan dikomposisi.
2. *Aspect*, dibuat jika tidak dapat dienkapsulasi dengan sempurna, bukan merupakan unit fungsional sistem, tetapi merupakan sebuah properti yang mempengaruhi unjuk kerja atau semantik *component* dengan cara yang sistematik.

Terminologi *component* yang dijelaskan di atas berbeda dengan pengertian *component* dalam paradigma sistem berbasis komponen.

Walaupun units *aspect* biasanya merupakan unit-unit NF, tidak tertutup kemungkinan untuk memanfaatkan kemampuan *aspect* untuk unit-unit fungsional.

Sebuah *aspect* sebagai unit modular dari sistem memiliki aturan komposisi sebagai berikut (34):

1. *Aspect* dapat mengimplementasikan interface atau mewarisi suatu kelas.
2. Kelas tidak dapat mewarisi *aspect*.
3. *Aspect* dapat mewarisi *aspect* lain.

Di dalam sebuah *aspect* terdapat beberapa elemen penting, yaitu: *joint point*, *advice*, dan *pointcut*.

Joint point adalah suatu lokasi pada urutan instruksi program yang diketahui dan terdefinisi. Dilokasi ini dapat disisipkan instruksi yang dapat mempengaruhi perilakunya. *Joint point model* menyediakan suatu kerangka referensi untuk melakukan definisi struktur aspek. *Joint point model* yang umum adalah pemanggilan suatu method (*method calls*). Contoh *joint point* yang lain adalah *exception*, *field definition* dan *access*. *Joint point model* yang disediakan tergantung bahasa PBA yang digunakan.

Advice adalah instuksi yang menyatakan perilaku yang akan disisipkan pada *joint point*. Pada kebanyakan bahasa PBA terdapat beberapa mekanisme untuk menjalankan *advice*, seperti: *before*, *after*, dan *around*.

Pointcut adalah sebuah konstuktor untuk mendefinisikan *joint point* yang akan digunakan. *Pointcut designator* mendeskripsikan sekumpulan *joint points*. *Pointcut designator* merupakan bagian penting dalam PBA karena menyediakan mekanisme untuk menyisipkan *advice* pada lebih dari satu *joint point* dalam program hanya dengan satu perintah. Mekanisme yang disediakan tergantung bahasa PBA yang digunakan.

3. HASIL DAN PEMBAHASAN

Pembangunan perangkat lunak berorientasi aspek adalah suatu cara yang sistematik untuk melakukan identifikasi, pemisahan, representasi dan komposisi *crosscutting concern* (13).

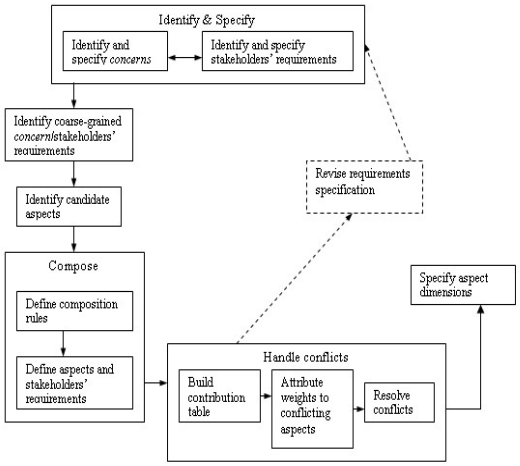
Tahapan proses yang dilalui tidak berbeda dengan tahapan pembangunan yang telah ada, sesuai dengan kondisi proyek atau aplikasi yang akan dibangun. Perbedaan terdapat pada metode yang digunakan. Metode yang digunakan sebaiknya juga sesuai dengan paradigma pemrograman (prosedural atau berorientasi objek) yang dipilih.

Perbedaan metode tersebut dibandingkan dengan metode yang tidak berorientasi aspek, terletak pada penekanan penanganan *concerns*, terutama *crosscutting concerns* untuk menemukan dan melokalisasi kebutuhan-kebutuhan aspektual.

Pada sub-bab 3.1. dan 3.2. dijelaskan dua metode identifikasi kebutuhan berorientasi aspek (*Aspects Oriented Requirement Engineering* – AORE), yaitu: *Modularisation and Composition of Aspectual Requirements* dan *Aspect-Oriented Requirement with UML*.

Pada sub-bab 3.3. dijelaskan metode analisis dan perancangan berorientasi aspek, yaitu: *The FRIDA Model*.

Pada sub-bab 3.4. dijelaskan usulan pengembangan notasi UML untuk tahap perancangan, yaitu: *Extending UML with Aspect: Aspect Support in the Design Phase*.



3.1. *Modularisation and Composition of Aspectual Requirements*

Metode ini memberikan pedoman untuk melakukan modularisasi dan komposisi kebutuhan aspektual dengan fokus modularisasi dan komposisi terhadap *concerns* kebutuhan yang berpotongan dengan kebutuhan lainnya pada tahap identifikasi kebutuhan. Sebagai contoh, kebutuhan penanganan keamanan tidak dapat dienkapsulasi menjadi satu buah *use case* atau *viewpoint*[5], tetapi biasanya tersebar pada beberapa *viewpoint* atau *use case.*

Dua motivasi diusulkannya pendekatan ini (28):

1. Menyediakan peningkatan dukungan untuk melakukan pemisahan properti fungsional dan non fungsional (yang bersifat *crosscutting*) pada tahap identifikasi kebutuhan. Untuk itu, disediakan cara identifikasi dan manajemen konflik yang disebabkan oleh *tangled representation*.
2. Mengidentifikasikan pemetaan dan pengaruh aspek dari tahap identifikasi kebutuhan terhadap ke tahap selanjutnya (setelah tahapan identifikasi kebutuhan).

Pendekatan ini dapat dilakukan dengan menggunakan teknik identifikasi kebutuhan (seperti *viewpoints*, *use case*, *goal-oriented*, *problem-frame*) dan bahasa yang digunakan untuk mendefinisikan spesifikasi kebutuhan adalah dalam format XML. Pada realisasinya, pendekatan ini menggunakan *viewpoints* (diadaptasi dari metode *viewpoint-oriented requirements engineering* yang disebut PREView[6]) dan XML. Realisasi pendekatan ini didukung dengan sebuah *tool* ARCaDe (*Aspectual Requirements Composition and Decision Suport Tool*). Model tersebut diusulkan untuk menangani *crosscutting concerns* pada tahap identifikasi kebutuhan seperti ditunjukkan pada gambar 8.

Gambar 8. Model AORE (28)

Model diawali dengan identifikasi dan spesifikasi *concerns* (aspek) dan kebutuhan fungsional. Dalam usulan tersebut, identifikasi kebutuhan dilakukan dengan menggunakan *viewpoints*. Identifikasi *concerns* dilakukan melalui analisis terhadap hasil identifikasi kebutuhan. Identifikasi kebutuhan maupun *concerns* boleh bersarang, dengan pengertian boleh terdapat sub kebutuhan maupun *sub-concerns*.

Setelah identifikasi kebutuhan dilakukan, didefinisikan keterkaitan antara kebutuhan dan *concerns* (*Identify coarse-grained concerns/ stakeholder requirement relationships*), dengan menggunakan matrik relasi *concern* dengan kebutuhan (lihat tabel 1). Dengan menggunakan tabel tersebut, ditentukan *concerns* yang menjadi kandidat *aspects*, yaitu *concerns* yang tersebar pada beberapa kebutuhan (SR – *stakeholder requirement*).

Tabel 1. Matriks Relasi *Concerns* Dengan Kebutuhan (28)

Stakeholder Req / *Concerns*	SR_1	SR_2	…	SR_3
$Concern_1$				√
$Concern_2$		√		√
…				
$Concern_3$	√	√		

[5] Finkelstein, A., Sommerville, I., (1996), *The Viewpoints FAQ*, BCS/IEE Software Engineering Journal, 11(1).

[6] Sommerviller, I., Sawyer, P., (1997), *Requirement Engineering – A Good Practice Guide*, Addison-Wesley.

Pada tahap *define composition rules* dapat digunakan spesifikasi *rules* meliputi *constrains* dan *outcome action*, serta *constrains* dan *outcome operator*. Spesifikasi *rules* dapat ditambah definisinya sesuai dengan konteks yang dibutuhkan. Dengan definisi aturan komposisi, dapat dispesifikasikan cara sebuah aspek mempengaruhi atau mengharuskan (*constrains*) perilaku dari sekumpulan SR. Kemudian dengan menggunakan *composition rules* tersebut, dilakukan komposisi antara aspek dengan kebutuhan.

Tabel 2. Matriks Kontribusi Antar Kandidat *Aspects* (28)

Aspects / Aspects	$Aspect_1$	$Aspect_2$	…	$Aspect_3$
$Aspect_1$		+		
$Aspect_2$				-
…				
$Aspect_3$				

Komposisi akan memberikan masukan pada identifikasi konflik antar kandidat *aspects*. Untuk menyelesaikan masalah tersebut (*handle conflicts*), dilakukan:

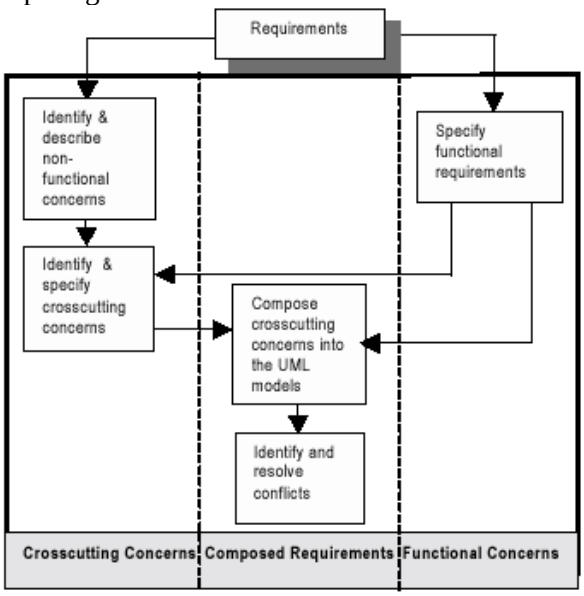
1. Pembuatan matrik kontribusi (lihat tabel 2), dengan memberikan tanda negatif (-) untuk kontribusi negatif, dan tanda positif (+) untuk sebaliknya. Konflik diidentifikasi dengan memperhatikan aspek yang memiliki kebutuhan yang sama ataupun yang tumpang tindih.
2. Pemberian ukuran bobot prioritas relasi kebutuhan dengan aspek yang mengalami kontribusi negatif dengan menggunakan matrik relasi *concern* dengan kebutuhan (lihat tabel 1).
3. Penyelesaian konflik bersama semua pihak yang terkait (*stakeholder* - pengguna, klien, dan lain lain) dengan menggunakan tabel bobot prioritas tersebut untuk membantu komunikasi.

Kegiatan terakhir adalah identifikasi dimensi aspek. Dimensi aspek dapat dikategorikan dalam 2 (dua) jenis (28):

1. *Mapping*: yaitu aspek yang akan dipetakan sebagai sebuah fungsi (method, objek, komponen), pilihan (arsitektur), rancangan atau implementasi *aspect*.
2. *Influence*: yaitu aspek yang mempengaruhi hal/ tempat yang berbeda dalam daur rekayasa (contoh: arsitektur, perancangan, spesifikasi)

3.2 *Aspect-Oriented Requirement with UML (AOR-UML)*

Model *AOR with UML* adalah adaptasi dari versi pertama (28) model AORE, yang dituangkan dalam model *Activity Diagram* seperti gambar 9.



Gambar 9. Model AOR-UML (2)

Model tersebut terbagi dalam 3 (tiga) bagian (2):

1. *Crosscutting concerns*: Identifikasi dan deskripsi kebutuhan non fungsional, termasuk identifikasi dan spesifikasi *crosscutting concern* dalam bentuk tabel 3.
2. *Functional concerns*: Identifikasi dan spesifikasi kebutuhan fungsional dengan menggunakan teknik yang telah ada (*traditional techniques*). Dalam usulan tersebut digunakan *use case* dan *sequence diagrams*.
3. *Composed requirements*: Komposisi kebutuhan fungsional dengan non fungsional, serta identifikasi dan pemecahan konflik yang timbul.

Tabel 3.Spesifikasi *Crosscutting Concerns* (2)

Crosscutting concern	<Name>
Description	<Executive description>
Priority	<Priority can be Max, Med and Min>
List of requirements	<Requirements that describe the concern>

List of models	<UML models influenced by the concern>

Komposisi FR dengan NFR mengadopsi konsep *overlapping*, *overriding* dan *wrapping* (2), yang dapat dijelaskan sebagai berikut:

1. *Overlapping*: Aspek melakukan modifikasi terhadap FR yang terkait. Aspek tersebut dapat berada pada titik sebelum atau setelah FR.
2. *Overriding*: Aspek merupakan *superpose* bagi FR yang terkait. Perilaku aspek tersebut menggantikan (*override*) perilaku FR.
3. *Wrapping*: Perilaku FR yang terkait dibungkus (*wrapped*) atau di-enkapsulasi (*encapsulated*) oleh aspek.

Dengan menggunakan konsep tersebut, hubungan komposisi digambarkan dengan menggunakan asosiasi antara *use case* FR dengan *use case* NFR. Asosiasi dan *use case* NFR digambarkan dengan menambahkan *stereotype*.

3.3. *The FRIDA Model*

FRIDA adalah singkatan dari *From RequIrement to Design using Aspects*. Fase-fase FRIDA terdiri dari:

1. *Requirements Identification*.
2. *Non Functional Requirements Refinement*.
3. *Lexicon Processing*.
4. *Conflicts Identification and Resolution*.
5. *Aspects Extraction*.
6. *Modeling FRs*.
7. *Visual Modeling of Aspects*.
8. *Combining Component and Aspects*.

Langkah pertama adalah analisis permasalahan sehingga dapat diidentifikasi kebutuhan. Selanjutnya dibentuk *use case diagram* yang memodelkan kebutuhan tersebut. Setiap *use case* dilengkapi dengan informasi seperti pada tabel 4.

Tabel 4. FR dan NFR *Template* (9)

Name	Descriptive phrase that names the use case.
Goal	Present the goal of this use case.
Author	Person responsible for elaborating the use case.
Pre-condition	The state in which the system will be before the use case begins.
Post-condition	The state in which the system will be after the use case has been completed.
Primary actor	The key actor in the use case.
Secondary actor	Another actor(s) that realizes any action.
Priority	Used to indicate how this use case will be delivered to customer.
Primary pathway	Description of the main event flow.
Alternate	Summarized description of the alternate event flow.
Exceptional	Summarized description of the exceptional flow of this use case.
Main	This section describes the main activities of the scenario of the use case. Observe that the focus here is on what must be done, and not how.
Variations	Here the steps that modify the main sequence steps are described.
Non-Functional	This section is reserved for appointing the generic NFRs related to the present use case: Performance: <resumed description> Security: <resumed description> Dependability: <resumed description>

Penghalusan NFRs dilakukan dengan menggunakan *checklists* untuk meyakinkan kelengkapan dari NFR, seperti contoh tabel 5.

Tabel 5. Contoh *Checklist* (9)

	R	N/A	P	Description
Dependability				
Reliability				
Is the system fault-tolerant?	O	O		
Does the system have exception handling or/and error recovery?	O	O		

R= relevan dengan konteks permasalahan, N/A= tidak beralasan, P= prioritas pada NFR terkait.

Selanjutnya dilakukan *Lexicon processing* atau pembuatan daftar kosa kata. Daftar kosa kata dibuat dengan notasi BNF[7] (*Backus Naur Form*) dan Leite[8] dengan menekankan pada kata-kata kunci yang terkait

[7] Naur, P., Backus, J. W., (1969), Programming systems and languages: *Revised Report on the Algorithmic Language Algol 60*", diubah oleh Saul Rosen, New York: McGraw-Hill.

[8] Leite, J.C.S.P., Oliveira, A.P.A., (1995), *A Client Oriented Requirements Baseline*, IEEE International Symposium on Requirements Engineering, IEEE Computer Society Press.

NFRs. *Lexicon processing* dibuat berdasarkan hasil deskripsi NFR dari *use case* dan *checklists*.

Identifikasi dan pemecahan konflik dilakukan dengan membuat matrik konflik antar aspek. Pemecahan dilakukan dengan negosiasi dengan pihak terkait.

Ekstraksi aspek dilakukan setelah *use case diagram* lengkap, dengan menggunakan bentuk tabel 6.

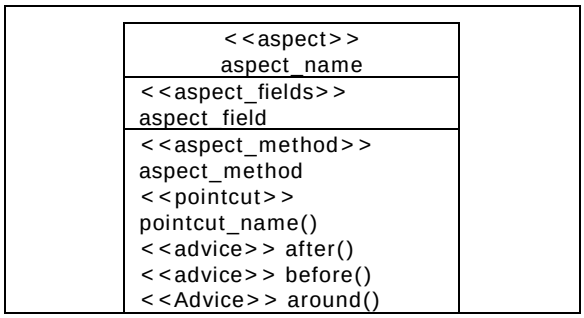
Tabel 6. *Template* Aspek (9)

Name	Descriptive word that names the aspect.
Level	It is used to indicate the present aspect level.
Description	The goal of the present aspect reported in this section.
Priority	Priority is used to indicate how aspect is important to the system under development.
Use cases list	This section of the template concentrates on enumerating the use cases related to this aspect.
Requirements list	Here the requirements of the present aspect are indicated.

Tingkatan (*level*) aspek diusulkan dalam 2 (dua) kategori, yaitu:

1. *Global*: Jika NFR terkait pada semua *use cases*.
2. *Partial*: Jika NFR terkait pada sebagian *use cases* saja.

Tahapan-tahapan selanjutnya merupakan tahapan perancangan. Perancangan FR dan NFR dilakukan secara terpisah. Perancangan FR dilakukan sebagaimana biasa dengan menggunakan *UML modeling*. Untuk perancangan NFR, notasi yang digunakan untuk memodelkan *aspect* adalah seperti gambar 10.

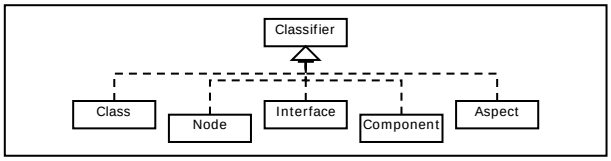


Gambar 10. Notasi *Class Diagram* Untuk *Aspect* (9)

3.4. Extending UML with Aspects: Aspect Support in The Design Phase

Usulan ekstensi UML[9] untuk mendukung perancangan perangkat lunak berorientasi *aspect* dibuat tanpa mengubah spesifikasi UML yang telah ada. Usulan tersebut mancakup usulan format dokumen untuk spesifikasi *aspect* dengan menggunakan XML untuk mendukung pertukaran informasi antar alat bantu rekayasa perangkat lunak (CASE *tools*). Format dokumen dibuat berdasarkan UXF (UML eXchange Format) dengan menggunakan ekstensi *aspect* yang diusulkan (UXF/a).

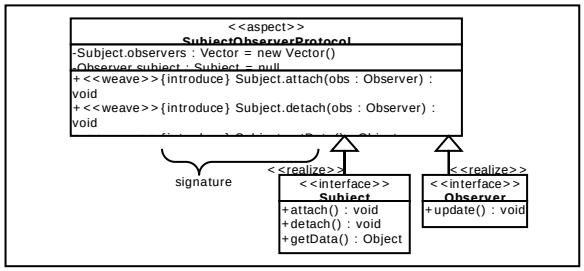
Notasi *aspect* diturunkan dari `classifier` (gambar 11), sehingga *aspect* juga dapat memiliki atribut, operasi dan relasi (*generalization*, *association*, dan *dependency*).



Gambar 11. *Aspect* Sebagai Metamodel Yang Diturunkan Dari *Classifier* (33)

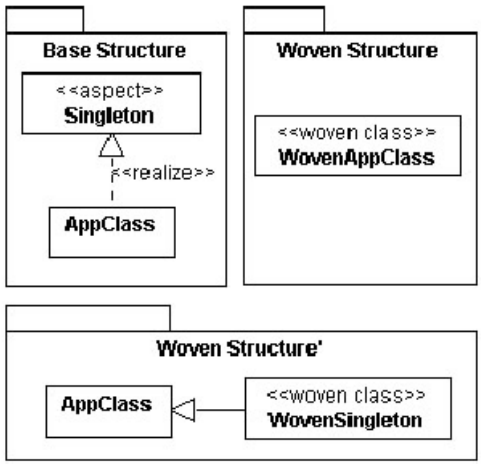
Aspect digambarkan dalam bentuk *class diagram* dengan *stereotype* <<aspect>> (gambar 12). Semua operasi pada *aspects* merupakan deklarasi *weaving*, diberi *stereotype* <<weave>>. Penanda *designator*[10] menggunakan *signature* (contoh: {introduce}).

Berdasarkan sifat dari *aspect*, relasi *aspect* dengan kelas merupakan turunan dari relasi *dependency* yaitu *abstraction* dengan *stereotype realization* (penggunaan relasi lihat gambar 12).



Gambar 12. Ilustrasi Notasi *Class Aspect* dan Relasi Aspect

Untuk menggambarkan struktur kelas yang telah bergabung dengan *aspects*, digunakan notasi kelas biasa dengan *stereotype* <<woven class>> (gambar 13). Untuk mendeskripsikan asal dari <<woven class>> (sebelum *aspect* dan kelas digabung), digunakan *tagged value* yang menunjukkan asal kelas dan aspek.



Gambar 13. Struktur *Woven Class* (33)

4. KESIMPULAN

1. *Separation of concern* (SOC) pada paradigma berorientasi aspek dilakukan pada golongan proses dan produk.
   a Prosesnya secara umum: identifikasi spesifikasi, komposisi, dan resolusi konflik.
   b Produknya secara umum: *non crosscutting concern* dan *crosscutting concern* (dapat berupa fungsional maupun non fungsional).
2. Pemisahan *crosscutting concern* dari *concern* yang tidak bersifat *crosscut* dilakukan sejak tahap identifikasi. Pemisahan masih dilakukan dengan cara masih mengandalkan intuisi, menggunakan *pattern*/ repositori, dan memperhatikan perilaku dari *concern*.
3. Dalam pengembangan perangkat lunak berorientasi aspek diyakini bahwa identifikasi aspek sedini mungkin akan memberikan kualitas perangkat lunak yang lebih baik. Oleh sebab itu, sebelum memasuki tahapan analisis, ditambahkan satu tahap identifikasi kebutuhan untuk melakukan identifikasi aspek sedini mungkin. Selain itu, pengembangan berorientasi aspek berusaha untuk memelihara SOC dari tahap konseptual hingga implementasi.

5. DAFTAR PUSTAKA

http://www.comp.lancs.ac.uk/computing/aop/, *Aspect-Oriented Software Engineering Special Interest Group*, Computing Department, Lancaster University, UK.

Araujo, J., Brito, I., Rashid, A., (2002), *Aspect-Oriented Requirement with UML*, Workshop on Aspect Oriented Modeling with UML.

Bakker, J., Tekinerdogan, B., Aksit, M., (2005), *Characterization of Early Aspects Approaches*, Department of Computer Science, University of Twente.

Baniassad, E., Clarke, Siobhan., (2004), *Finding Aspects in Requirements with Theme/Doc*, Workshop Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Lancaster University, UK.

Bar-On, D., (2004), *Early Aspects – Overview of Some Approaches*.

Bergey, John., dkk, (2004), Technical report: *Result of SEI Independent Reseach and Development Projects and Report on Emerging Technologies and Technology Trends*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA.

Bramanian S., (2003), *Pembangunan Simulator Mekanisme Static Weaving pada Pemrograman Berorientasi Aspek*, Skripsi Program Sarjana, Institut Teknologi Bandung, Bab 2, 1 - 24.

Brito, I., Moreira, A., (2004), *Integrating the NFR framework in a RE model*, Workshop