Pengantar Pemrograman Berbasis Aspek *(AOP)*

Adhari C Mahendra

elnuur@yahoo.com
aop@groups.or.id

Abstrak:
Paradigma rekayasa perangkat lunak terakhir yaitu berbasis obyek (*object-oriented*) telah meraih sukses besar. Usaha meningkatkan mutu desain program dengan dicapainya peningkatan readability, reusability serta dekomposisi struktural (*modularity*). Ini merupakan motivasi utama pergeseran dari pemrograman prosedural menjadi berbasis obyek. Dalam desain berbasis obyek, lokalisasi komponen dan obyek didasarkan atas unit fungsi (seperti book, account, log). Satu desain unit fungsi yang bersih, besar kemungkinan akan dimodifikasi, untuk menambahkan fitur yang melibatkan berbagai macam unit fungsi lain. Sebagai contoh unit fungsi *account*, untuk mendapatkan unit fungsi *tracing*, *profiling*, atau *auditing*, maka ditambahkan unit fungsi *Log*. *Scattering* (satu *concern* muncul di mana-mana) dan *tangling* (satu obyek/komponen terdapat berbagai macam *concern*) cenderung muncul bersamaan, sebagai konsukuensi logis dari adanya berbagai macam fitur dalam satu unit fungsi. Hal inilah yang menjadikan desain berbasis obyek tidak lagi memiliki modularitas yang bersih. Muncul ide membentuk paradigma baru yaitu berbasis aspek (*aspect-oriented*). Satu aspek mewakili satu unit fungsi (*concern*) yang bisa menjadi fitur di unit fungsi lain.

Keywords: aop, aspect-oriented, programming, reflection, advice, pointcut, joinpoint, weaving, crosscut, aspectj

1. PENDAHULUAN

Paradigma berbasis obyek telah membawa dunia rekayasa perangkat lunak menapaki era baru. Bahkan boleh dikatakan sebagai babak revolusi dalam pengembangan perangkat lunak. Ide besar *Design Pattern* yang dipromotori oleh 'Gang of Four' (GoF, Erich Gamma, Richard Helm, Ralph Johnson, dan John Vlissides)

tampil pada era ini. Berbagai macam metodologi baru dalam pengembangan perangkat lunak pun mulai marak, salah satu yang paling populer adalah *Unified Software Development Process* (aka RUP) dengan *Object-Oriented Analysis and Design* (OOAD)-nya yang dimotori oleh Grady Booch, Ivar Jacobson, dan James Rumbaugh. Notasi-notasi yang merupakan ujung tombak dari *Computer Aided Software Engineering* (CASE) pun banyak muncul ke permukaan. Seperti notasi Booch, OMT, dan yang paling populer belakangan ini adalah *Unified Modeling Language* (UML).

Orientasi obyek menawarkan tiga konsep dasar dalam pemrograman:

- *Encapsulate* (Enkapsulasi)
- *Inheritance* (Penurunan)
- *Polymorphic*

Ketiga konsep inilah yang menjadi landasan kuat bagi pemrograman berbasis obyek sehingga dapat meningkatkan *readability* (mudah dibaca dan dipelajari), *reusability* (mudah digunakan kembali), dan dekomposisi permasalahan (*modularity*) berdasarkan obyek atau fungsi.

Pada umumnya, pengembangan aplikasi di dunia bisnis –*Enterprise Application Development* (EAD)- yang berbasis obyek membagi aplikasi menjadi beberapa *layer* (lapis) berdasarkan obyek dan fungsinya. Secara sederhana, minimal terdapat *Persentation Layer* yang menangani masalah *user interface* dan antar muka dengan aplikasi lain, *Business Objects Layer* yang merupakan inti kerja dari suatu aplikasi, dan *Persistance Layer* yang menangani antar muka data ke sistem *database*.

Business Object merupakan unit representasi dari model bisnis yang dijalankan oleh aplikasi, seperti Account dan CreditTransfer. Selain dari konsrentasi bisnis (*business concerns*), supaya aplikasi lebih mudah dipantau dan dikelola, diberikan fitur-fitur lain, seperti *Logging* atau *Tracing*, *Security*, *Session*, *Pooling*, dan lain-lain, yang kesemuanya itu bukan konsentrasi bisnis, namun merupakan kensontrasi teknis (*technical concerns*). Hal ini mengakibatkan *source code Business Object* tidak benar-benar bersih, karena di dalamnya ada aspek-aspek lain dari obyek bisnis itu sendiri, ini disebut *tangling*. Dari sisi teknis *Logging* misalnya, kode program yang mengakses *Logging* ini bisa muncul di berbagai macam kode program *business objects* bahkan juga ada di setiap layer aplikasi, hal ini disebut *scattering*. *Scattering* dan *tangling* ini cenderung akan muncul secara bersamaan dalam pengembangan perangkat lunak. Menyebabkan *source code* menjadi tidak independen dan bersih.

Pada tahun 1996, Gregor Kickzales, berangkat dari ide *reflection* dalam pemrograman, melontarkan gagasan brilian tentang pemrograman berbasis *aspect* (*aspect-oriented programming*, AOP). Pada tahun 1998-2000, Gregor Kickzales mendapat grant dari DARPA untuk melakukan riset tentang AOP yang terbagi menjadi 2 fase. Fase pertama 1998-1999 lebih banyak tentang AOP itu sendiri sebagai landasan, sementara fase kedua 2000-2002 adalah pengembangan AspectJ sebagai implementasi AOP. Walaupun pada masa awal ide ini boleh dikatakan tidak terlalu banyak mendapat dukungan, namun sejak tahun 2000, melalui usaha tiada henti dan memberikan contoh yang lebih nyata dengan implementasi AspectJ, akhirnya publik pun mulai menunjukkan penerimaannya terhadap gagasan ini. *Project* dan implementasi pemrograman berbasis *aspect* ini pun mulai marak dalam berbagai bahasa. AspectC merupakan *project* implementasi *aspect-oriented* dengan bahasa C/C++, AspectS untuk Smalltalk, AspectNet untuk C# .Net, Pythius untuk Phyton, dan AspectR untuk Ruby. Sementara AspectJ, HyperJ, DemeterJ, Java Aspect Component (JAC dari AOPSYS), Nano, AspectWerkz, dan lain sebagainya, merupakan jajaran nama-nama *project* implementasi *aspect-oriented* untuk Java. Sementara itu JSR 1.5 'Tiger', memasukan ide *metadata* yang merupakan komplementari dari *aspect-oriented*.

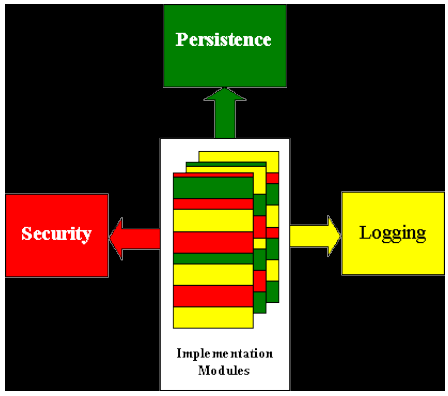
Pemrograman berbasis aspek (*aspect-oriented programming*) hadir untuk memotong silang (*crosscut*) permasalahan *scaterring* dan *tangling*. Metodologi ini bukan menggantikan metodologi pemrograman berbasis obyek atau pendahulunya, namun hanya merupakan ekstensi pengembangan dari yang ada.

2. CROSSCUTTING CONCERNS

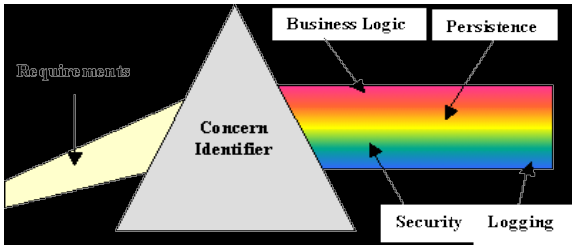
Prilaku *crosscutting concerns* bisa berbagai macam cara. Seorang *developer* membuat suatu sistem berdasarkan berbagai macam *requirements* (kebutuhan). Kita bisa mengklasifikasikan *requirement* ini secara luas menjadi kebutuhan *level* inti modul dan kebutuhan *level* sistem. Banyak kebutuhan *level* sistem cenderung menjadi ortogonal (independen secara mutual) terhadap yang lain dan juga ke kebutuhan *level* modul. Kebutuhan *level* sistem cenderung untuk me-crosscut banyak inti modul. Sebagai misal *Session* sebagai kebutuhan *level* sistem akan ortogonal dengan kebutuhan *level* sistem yang lain misal *Security*, dan juga ortogonal dengan kebutuhan *level* modul misal *CreditTransfer*.

Walaupun *crosscutting concerns* muncul di berbagai macam modul, teknik implementasi sekarang ini cenderung untuk mengimplementasikan *requirements* ini mengunakan metodologi satu dimensi. Implementasi satu dimensi ini cenderung hanya mengimplementasi kebutuhan *level* modul saja. Kebutuhan *level* lainnya hanya sekedar ditambahkan di sepanjang dimensi yang dominan ini. Dengan kata lain, kebutuhan merupakan dimensi ruang n (*n-space*), sementara implementasi berdimensi tunggal. Tampaklah bahwa pemetaan kebutuhan ke implementasi tidak memiliki hasil yang cocok.

Kita bisa melihat suatu sistem yang kompleks sebagai kombinasi implementasi dari berbagai *concern*. Sistem yang kompleks terdiri dari berbagai macam concern, seperti *business logic*, *performance*, *data persistence*, *logging*, *debugging*, *authentication security, multithreaded safety*, *error checking*, dan lainnya. Bahkan seringkali kita menemui *concern* yang berhubungan dengan proses *development*, seperti *comprehensibility*, *maintainability*, *tracebility*, dan kemudahan evolusi bagi kode program. Gambar berikut ini menggambarkan suatu sistem sebagai himpunan *concern* diimplementasikan dalam berbagai macam modul.



Gambar berikutnya menunjukkan suatu himpunan kebutuhan sebagai sinar cahaya melewati suatu prisma. Kita lewatkan sinar cahaya kebutuhan melewati prisma penentuan *concern* yang memilah tiap *concern*.



AOP melakukan *corsscuting concern* sebagaimana OOP telah melakukan enkapsulasi (*encapsulation*) dan penurunan (*inheritance*) obyek yang dilengkapi mekanisme bahasa pemrograman yang secara eksplisit mampu untuk menangkap struktur dari *crosscutting* ini. Hal ini memungkinkan untuk memprogram *crosscutting concern* ini dengan cara yang modular, dan mendapatkan keuntungan umum yaitu peningkatan modularitas. Kode program menjadi lebih sederhana yang artinya lebih mudah di kembangkan dan dikelola, dan memiliki potensi lebih besar untuk digunakan kembali. Modularisasi *crosscutting concern* dinamakan aspek (*aspect*).

Sebagai contoh, kebutuhan spesifik untuk *security* dalam satu aplikasi, yang tidak dirancang untuk concern aplikasi ini, akan mengenai beberapa komponen atau class. Dalam kasus ini, security bisa dianggap sebagai satu *aspect* dalam AOP, tanpa menggangu desain sistem aslinya.

Coba pertimbangkan contoh sederhana berikut namun lebih nyata. Pertimbangkan kerangka implementasi dari sebuah *class* menenkapsulasi suatu *business logic*.

```
public class SomeBusinessClass extends OtherBusinessClass {

    // Core data members
```

```
    // Other data members: Log stream, data-consistency
flag
   // Override methods in the base class
    public void performSomeOperation(OperationInfo info) {
       // Ensure authentication
        // Ensure info satisfies contracts
         // Lock the object to ensure data-consistency in
case other
        // threads access it
       // Ensure the cache is up to date
        // Log the start of operation
       // ==== Perform the core operation ====
         // Log the completion of operation
        // Unlock the object
    }
     // More operations similar to above
    public void save(PersitanceStorage ps) {
    }
    public void load(PersitanceStorage ps) {
    }
}
```

Tampak kerangka *class* di atas memiliki banyak *concern* yang bukan bagian dari *business logic*. Di dalam suatu *method* terdapat *concern* diluar *business logic*, seperti *authentication*, *locking*, *thread safety*, dan lainnya, juga dalam satu *class business logic* ada *method* yang diluar *concern business logic*, seperti *persistence* (*method save* dan *load*).

Walaupun AOP memberikan kemampuan untuk *crosscutting concern* tak terlihat secara sekilas, beberapa *aspect* bisa saja muncul beberapa kali pada masa pengembangan perangkat lunak, dan dapat diintegrasikan pada awal desain. Secara garis besar *aspect* dapat dikategorisasikan menjadi *development aspect* dan *production aspect*.

3. DEVELOPMENT ASPECT

Development aspect digunakan selama proses *development* dan diharapkan untuk dihilangkan pada aplikasi yang final. *Aspect* ini dapat diaktifkan atau dinon-aktifkan pada tiap beda tahapan dari proses *development*.

Tracing, *profiling*, *counting*, dan *logging* merupakan *aspect* yang dapat dijadikan contoh. Seringkali *programmer* menambahkan *trace statements* ke dalam kode program dan dihilangkan ketika *debugging* selesai. Mengganti *concern* ini dengan *aspect* dapat membantu pemisahan instruksi ekstra ini lebih baik.

Banyak *programmer* menggunakan model 'Design by Contract' yang dipopulerkan oleh Bertand Meyer. Dalam model pemrograman ini *pre-* dan *post-conditions* pemanggilan suatu *method* diberikan secara eksplisit agar dapat berjalan sesuai dengan yang diharapkan. *Pre-* dan *port-conditions* dapat diimplementasikan sebagai *aspect*. *Aspect* dapat digunakan untuk memastikan *method* tertentu dipanggil dengan parameter yang benar dan memberikan dan/atau mengembalikan kembalian (*return*) yang benar. Sebagai contoh, dalam suatu sistem yang berhubungan dengan pengukuran jarak, sangat penting untuk memastikan nilai masukan yang diberikan ke komponen perhitungan selalu bernilai positif guna menjaga sisyem berjalan konsisten. *Aspect* semacam ini didapati sampai aplikasi yang final.

Mekanisme *crosscut* berbasis *property* dapat berguna dalam mendefinisikan pelaksanaan kontrak yang rumit. Satu kekuatan besar dari penggunaan mekanisme ini adalah untuk mengidentifikasi pengambilan dan penulisan property melalui pemanggilan method. Hal semacam ini akan menjaga konsistensi antara desain dan implementasi juga ketika *compile* dan *run-time*.

Selain itu juga untuk menjaga konsistensi desain *layer*, sebagai contoh, tidak diperkenankannya *Persentation Layer* langsung mengakses *Persistance Layer*, *aspect* dapat membantu untuk menjaga hal seperti ini ketika *compile-time*. *Aspect* semacam ini bisa dihilanggkan ketika *run-time*.

4. PRODUCTION ASPECT

Production aspect lebih digunakan dalam konteks yang bersifat operasional, yaitu ketika user menjalankan aplikasi final.

Java beans merupakan komponen perangkat lunak yang *reusable* yang secara visual dapat dimanipulasi oleh *tool* IDE. Sedikit *requirement* menjadikan obyek menjadi *sebuah* bean. *Bean* harus didefinisikan dengan konstruktor tanpa

argument dan harus diturunkan dari Serializable atau Externalizable. *Property* yang ada di obyek diperlakukan sebagai *property bean* yang harus dindikasikan dengan adanya method *get* dan *set* yang penamaannya *getProperty* dan *setProperty* dimana property merupakan nama *field* dalam *class bean*. Beberapa *property* dari *bean*, yang dikenal dengan *bound property*, mengeluarkan *event* ketika terjadi perubahan nilai property, sehingga *listener* yang terdaftar akan dapat informasi perubahan tersebut. Dalam membuat *bound property*, melibatkan penyimpanan daftar *listener*, serta pembuatan dan *dispatch event* obyek di method yang nilai *property*-nya berubah. *Dispatch event* seperti ini biasa disebut sebagai *change monitoring*. *Change monitoring* yang akan men-*crosscut* dekomposisi fungsional dapat digunakan untuk meningkatkan penanganan *persistence* dan *cache*.

Crosscuting struktur dari *context passing* bisa menjadi sumber kompleksitas utama pada bahasa Java. Pertimbangkan implementasi fungsionalitas yang memperbolehkan *client* dari suatu editor untuk merubah warna ke suatu *element* gambar. Biasanya ini membutuhkan *passing* suatu warna, atau *color factory*, dari *client* menuju ke pemanggilan suatu *method* di *element factory*. Semua programer biasa menghadapi ketidak-nyamanan dalam menambahkan satu argumen ke *method-method* yang berhubungan, hanya karena untuk melakukan *passing* informasi semacam ini. Dengan menggunakan *aspect*, *context passing* semacam ini dapat diimplementasikan dengan cara yang modular.

Property-based aspect dapat digunakan untuk menjaga konsistensi penanganan fungsionalitas dari banyak operasi. Seperti untuk memastikan *log* setiap ada *exception* terjadi.

Banyak *development* pada suatu aplikasi membutuhkan perubahan kecil pada *requirement* yang disesuaikan dengan kebutuhan permasalahan bisnis *client*. Hal ini cenderung sedikit merubah fungsionalitas pada satu atau dua *method*. AOP dapat menjawab isu ini daripada mengimplementasi ulang *class* atau menurunkan (*inherits*) dan meng-*override*-nya dari *class* induknya.

Aspect dapat membantu *design pattern* yang menggunakan *class helper* yang membutuhkan banyak penambahan code untuk menangani kebutuhan *level* sistem. *Synhconous*, *distribution* (*load balancing*), *resource pooling*, *storage management*, *session management*, *performance*, *authentication*, dan administrasi merupakan contoh lain dari kebutuhan *level* sistem yang ada di *production* untuk aplikasi berskala *enterprise*. Setiap *aspect* men-*crosscut* berbagai sub-sistem, sebagai contoh, *storage management* mengenai setiap *business object* yang *stateful*.

5. DASAR AOP

Para peniliti telah mempelajari berbagai macam cara untuk menyelesaikan masalah dibawah topic umum dari '*separation of concern*'. AOP merepresentasikan salah satu metode tersebut.

AOP, pada intinya, menyediakan implementasi *individual concern* dengan cara yang *loosely coupled*, dan mengkombinasikan implementasi ini untuk membentuk sistem yang final. AOP membuat sistem menggunakan *loosely coupled* dan modularisasi implementasinya dari *cosscutting concern*. OOP, sebaliknya, membuat sistem menggunakan *loosly coupled* dan modularisasi implementasinya dari *common concern*. Unit modularisasi dari AOP disebut sebagai *aspect*, dan implementasi *common concern* di OOP disebut *class*.

AOP melibatkan tiga tahap development:

- **Aspectual decomposition**

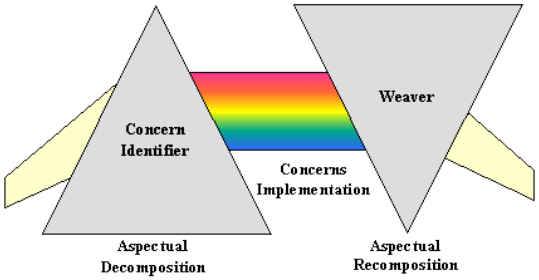
  Memisahkan kebutuhan untuk mengidentifkasi *crosscutting* dan *common concern*. Pemisahan *concern level* modul dari *crossutiing concern level* sistem. Sebagai contoh *credit card* modul, dapat didentifikasi menjadi 3 *concern*: inti pemrosesan *credit card*, *logging*, dan *authentication*.

- **Concern implementation**

  Implementasi tiap *concern* secara terpisah. Untuk contoh *credit card*, diimplementasikan unit inti pemrosesan *credit card*, unit *logging*, dan unit *authentication*.

- **Aspectual re-composition**

  Pada tahap ini *aspect integrator* menspesifikasikan aturan rekomposisi dengan membuat unit modularisasi yaitu *aspect*. Proses rekomposisi juga dikenal sebagai *weaving*, menggunakan informasi ini untuk membentuk sistem yang final.



AOP banyak berbeda dengan OOP dalam hal menentukan *crosscutting concern*. Dengan AOP, tiap implementasi *concern* tidak perlu memperhatikan *concern* yang lain. Sebagai contoh modul pemrosesan *credit card* tidak perlu mengerti *concern* yang lain seperti *logging* atau *authentication* dan bagaimana operasinya. Ini merepresentasikan pergeseran paradigma yang besar dari OOP.

Suatu implementasi AOP dapat diberlakukan pada metodologi pemrograman yang lain sebagai metodologi dasarnya yang membawa keuntungan sistem dasarnya. Sebagai contoh implementasi AOP bisa mengambil OOP sebagai sistem dasar untuk mendapatkan keuntungan implementasi *common concern* dengan OOP. Dengan implementasi tersebut, tiap *concern* menggunakan OOP untuk mengidentifikasi tiap *concern*. Ini analog dengan bahasa prosedural yang bertindak sebagai dasar bahasa untuk berbagai bahasa OOP.

6. ANATOMI BAHASA AOP

Seperti halnya metodologi pemrograman yang lainnya, implementasi AOP terdiri dari dua bagian: spesifikasi bahasa dan implementasi. Spesifikasi bahasa menjelaskan bagaimana suatu bahasa dibangun dan juga *syntax*-nya.. Implementasi memverifikasi kode program berdasarkan spesifikasinya dan merubahnya menjadi bentuk yang dapat dieksekusi di mesin tujuan, umumnya dilakukan oleh *compiler* atau *interpreter.*

Pada level yang lebih tinggi, suatu bahasa AOP menspesifikasikan dua komponen:

- **Implementation of concerns**

  Pemetaan masing-masing kebutuhan manjadi kode program sehingga *compiler* dapat merubahnya menjadi kode yang dapat dieksekusi oleh mesin. Karena implementasi *concern* mengambil bentuk dari prosedur, maka bahasa tradisional seperti C, C++, atau Java dapat digunakan dengan AOP.

- **Weaving rules specification**

  Bagaimana menyusun secara independen *concern-concern* yang telah diimplementasikan ke bentuk sistem yang final. Untuk mencapai tujuan ini, implementasinya dapat menggunakan atau membuat bahasa untuk menentukan aturan-aturan penyusunan pecahan implementasi yang berbeda tersebut ke dalam bentuk sistem yang final. Bahasa yang menentukan aturan *weaving* bisa hanya merupakan ekstensi dari bahasa yang ada atau bisa berupa bahasa yang baru atau berbeda sama sekali.

Compiler bahasa AOP melakukan dua tahapan logika:

- Menggabungkan masing-masing *concern*
- Merubah informasi hasilnya menjadi kode yang dapat dieksekusi.

Suatu implementasi AOP dapat mengimplementasikan *weaver* dalam berbagai cara, termasuk translasi dari *source code* ke *source code*. Di sini pertama kita menyiapkan *source code* untuk tiap *aspect* untuk menghasilkan *source code* yang di-*weaving*. Kemudian *compiler* AOP menaruh kode hasil konversi ini ke *compiler* bahasa dasarnya untuk menghasilkan kode final yang dapat dieksekusi. Sebagai contoh yang menggunakan pendekatan ini, suatu implementasi AOP berbasis Java akan

menkonversi tiap-tiap *source code aspect* ke *source code* Java, kemudian Java *compiler* merubahnnya menjadi *byte-code*. Pendekatan yang sama juga dapat dilakukan untuk *level byte-code*, bagaimanapun *byte-code* juga merupakan bentuk lain dari *source code*. Selain itu, sistem eksekusi –katakanlah VM- dapat dibuat untuk mengenali *aspect*. Dengan pendekatan ini, untuk implementasi AOP berbasis Java misalnya, VM pertama-tama akan mengambil aturan-aturan *weaving*, dan memberlakukannya untuk setiap pemanggilan *class-class* berikutnya. Dengan kata lain, ini merupakan *just-in-time weaving*.

7. PENDEKATAN DAN IMPLEMENTASI AOP

Desain sistem berbasis aspek (*Aspect-Oriented System Design*, AOSD) masih dalam tahap yang terlalu dini digunakan dalam *development* juga terminologinya, sebagai mana konsepnya sendiri masih pelan-pelan dikembangkan dan dipersatukan.

Walaupun demikian, beberapa grup riset telah mengembangkan implementasinya masing-masing, konsep utama AOSD hanya berbagi akar yang sama. Sebagian besar didefinisikan dalam [KLM+97], namun terdapat perbedaan pengartian ketika digunakan dalam konteks suatu teknik AOP tertentu.

Implementasi AOP secara umum dari suatu aplikasi dapat dibagi menjadi tiga bagian:

- **Component program**

  Bagian aplikasi ini merepresentasikan dekomposisi fungsional. Ini diprogram dalam bahasa komponen, seperti Java, atau juga suatu bahasa spesifik aplikasi.

- **Aspect Program**

  Bagian aplikasi ini merepresentasikan *crosscut* dengan unit fungsional yang terdapat dalam komponen program. Ini dapat ditulis dalam suatu bahasa spesifik aspek (seperti AspectJ) atau ditentukan dengan mekanisme konfigurasi dengan pola tertentu (seperti JAC).

- **Weaving**

  Bagian ini merubah komponen program dan aspek program menjadi sistem yang final. Ini mengintegrasikan instruksi ekstra yang dihasilkan untuk aspek kedalam desain komponen. Ini bisa dilakukan pada *compile-time* (AspectJ atau *run-time* (JAC).

Konsep kunci dari join-point didefinisikan sebagai "*[an element] of the component language semantics that the aspect programs coordinate with*". Ini menggunakan terminologi AspectJ, merupakan titik yang dapat ditentukan dalam eksekusi program. Ini memungkinkan untuk membedakan dua percabangan dengan bahasa aspek dihubungkan ke bahasa komponen.

Di satu sisi, beberapa implementasi AOP menggunakan bahasa pemrograman yang sudah ada untuk komponennya. Sebagai contoh, bahasa komponen untuk AspectJ adalah Java. Ini juga mungkin untuk membuat ekstensi untuk bahasa pemrograman yang sudah ada seperti C, Smalltalk, atau Ruby.

Di sisi lain, beberapa implementasi melibatkan bahasa komponen khususnya dibuat untuk tujuan fungsional dari aplikasi. Pasti lebih baik menenkapsulasi *concern* fungsional dengan *application-specific-construct.*

8. CONTOH IMPLEMENTASI AOP DENGAN ASPECTJ

Untuk lebih jelasnya coba, kita akan melihat gambaran dengan menggunakan kerangka source code. Pada contoh ini, komponen program ditulis dalam bahasa Java, sedangkan aspek program ditulis dalam AspectJ. Di sini kita tidak akan mengupas bahasa AspectJ, mungkin hanya sedikit yang dikupas untuk tujuan agar bisa dipahami. Bahasan tentang AspectJ akan dikupas dalam tulisan lain.

Perhatikan contoh kerangka program berikut dan bandingkan dengan contoh kerangka program sebelumnya.

```
public class SomeBusinessClass
```

```
extends OtherBusinessClass {
    // Core data members
    // Override methods in the base
class
                    public     void
 performSomeOperation(OperationInfo
info) {
          // ==== Perform the core
operation ====
    }
}
```

Kerangka program diatas menunjukkan kode program yang bersih dan hanya memiliki concern yang berhubungan dengan *class* tersebut saja. *Concern-concern* lain seperti *log*, *authentication*, *synchronatization*, *object locking*, *multithreaded safety*, dan *persistence* tidak lagi dimuat di sana.

Berikut ini kita akan membuat *aspect* yang akan men-*crosscut class* tersebut dengan *concern Log* sederhana berikut ini.

```
public aspect LogAspect {
        pointcut traceMethods() :
(execution(* *.*(..))
                || execution(*.new
(..))) && !within(LogAspect);
    before() : traceMethods() {
                Signature sig =
thisJoinPointStaticPart.getSignatur
e();
       System.out.println("Entering
["
             + sig.getDeclaringType
().getName() + "."
           + sig.getName() + "]");
    }
    after() : traceMethods() {
                Signature sig =
thisJoinPointStaticPart.getSignatur
e();
        System.out.println("Leaving
["
             + sig.getDeclaringType
().getName() + "."
           + sig.getName() + "]");
    }
}
```

Dari contoh tersebut ada tiga bagian yang harus dipahami, yaitu:

- **Aspect**

  Merupakan komponen utama dari suatu aspek yang merepresentasikan satu *concern*. Aspek direpresentasikan dengan *aspect identifier*, yang setara dengan *class identifier*. Syntaxnya:
  <access specifier> [abstract] aspect <aspect-name> [extend <class-or-aspect-name>] [implements <interface-list>] { < aspect-body> }
  Contoh (diambil dari contoh diatas):
  public aspect LogAspect { … }
  yang menyatakan *aspect* bernama LogAspect dan memiliki *access specifier* bertipe *public*.

- **Pointcut**

  Merupakan titik pemotongan ke komponen program. Ini menjelaskan kapan, dimana, dan bagaimana suatu pemotongan terjadi. Bisa dikatakan ini seperti trapping komponen program untuk ditangkap dan dilempar ke aspek program. Syntax:
  [access specifier] pointcut <pointcut-name([args]) : pointcut-definition.
  Pointcut-definition dapat dinyatakan sebagai himpunan pernyataan logika.
  Contoh:

```
pointcut traceMethods() :
(execution(* *.*(..))
          || execution(*.new
(..))) && !within(LogAspect);
```

  Nama *pointcut* diatas adalah traceMethods tanpa argumen. *Pointcut-definition*-nya adalah setiap eksekusi *method* di semua *class*, atau setiap eksekusi pembuatan *instance* baru semua *class* (*constructor*) namun tidak dari kode program yang ada di aspek LogAspect sendiri.

- **Advice**

  Advice sangat berhubungan erat dengan *pointcut*, bisa didefinisikan oleh *pointcut* seperti diatas maupun oleh *advice* itu sendiri. *Advice* merupakan bagian eksekusi kode program yang ada di *body* dari aspek, tampak seperti *body* dari suatu *method* dalam suatu *class*. Tipe advice ada 3: ***before*** yaitu titik sebelum *pontcut*

terjadi, ***around*** yaitu titik sebelum dan sesudah yang ditandai dengan eksekusi kode **proceed(…)**, dan ***after*** yang merupakan titik setelah pointcut. *After* sendiri terbagi menjadi 3: ***after returning*** yaitu *after* yang kembali dengan normal tanpa *exception*, ***after throwing*** yaitu *after* dari adanya *exception*, dan *after* sendiri yang menangkap semua kembalian baik itu normal atau dengan *exception*. Syntax:
<advice-type> <advice-name([args])> [returning-or-throwing-for-after-adviced([args])] : <pointcut-name | pointcut-definition>

Dengan AspectJ *compiler*, contoh komponen program dan aspek program di atas akan di-*weave* menjadi kode program kira-kira seperti berikut ini.

```
public class SomeBusinessClass extends OtherBusinessClass {
    // Core data members
    // Override methods in the base class
                    public void performSomeOperation(OperationInfo info) {
          //  body from LogAspect before adviced
          // ==== Perform the core operation ====
           // body from LogAspect after adviced
    }
}
```

Dengan mengimplementasikan aspek-aspek yang lain kita akan mendapatkan hasil kode program yang di-*weave* berjalan sesuai dengan *requirements*.

9. KESIMPULAN

Pada makalah ini telah dibahas tentang pengenalan dari pemrograman berorientasi aspek (AOP) dan juga dilengkapi ilustrasi contoh dari salah satu implementasinya dengan menggunakan AspectJ. Salah satu tujuan dari makalah ini adalah untuk memberikan gambaran secara global dari AOP juga jawaban atas tantangan dari rekayasa perangkat lunak yang menggunakan metode berbasis obyek atau yang lebih lama. Namun, kesemuanya belumlah lengkap jika kita tidak memiliki gambaran obyektif dalam implementasi dari AOP ini. Untuk itu berikut ini akan dikupas kelebihan dan kekurangan dari AOP ini.

Kelebihan:

- Tanggung jawab yang lebih bersih dari tiap modul
- Modularisasi yang lebih baik
- Mudahnya sistem untuk berevolusi
- Keputusan desain *late-binding*
- Kode lebih *reusable*

Kekurangan:

- Alur program di AOP sulit untuk ditelusuri tidak seperti di OOP atau bahkan di procedural. Dengan jumlah concern yang sedikit, otak kita masih bisa menulusurinya, namun ketika jumlah concern menjadi banyak sangat sulit untuk melakukan penelusuran. Sebenarnya pada OOP juga terjadi hal demikian, *polymorphic* method membuat analisa alur program menjadi rumit, begitu juga dengan bahasa prosedural seperti C, adanya function pointer, *flow* program menjadi tidak statis dan sulit dipahami.
- AOP tidak menjawab semua masalah baru itu benar, ia hanya menjawab permasalahan yang sebelumnya tidak terpecahkan dengan cara yang lebih baik dengan lebih sedikit *effort* dan meningkatkan *maintainability*-nya. Kita bisa menyelesaikan semua permasalahan dengan metodologi apapun, namun yang membedakan adalah kompleksitas solusinya. Pada kenyataanya tidak ada yang tidak dapat diimplementasikan menggunakan bahasa mesin.

- AOP melanggar encapsulation namun hanya secara semantik dan kontrol. Pada OOP, *class* menenkapsulasi semua behavior. Hasil *weaving* dari AOP menghapus *level* kontrol ini. Dapat dikatakan AOP menawarkan kekuatan yang dapat dipertimbangkan, dan ini dapat bekerja mengagumkan jika kita menggunakan kekuatan itu dengan benar.
- AOP membutuhkan desain yang bagus dari komponen program sebagai inti *concern*.

10. *POINTER* dan *RESOURCE* MENGENAI AOP

AspectJ *www.aspectj.org*

JAC *jac.aopsys.com*

DemeterJ *www.ccs.neu.edu/demeter/dj*

I want My AOP *www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2-p3.html*

Aspect Refactoring *www.theserverside.com/resources/article.jsp?l=AspectOrientedRefactoringPart*

11. DAFTAR SINGKATAN

AOP = Aspect-Oriented Programming

OOP = Object-Oriented Programming

AOSD = Aspect-Oriented Software Design

CASE = Computer Aided Software Engineering

JAC = Java Aspect Component

OOAD = Aspect-Oriented Analysis and Design

OMT = Object Modelling Technique

RUP = Rational Unified Process

UML = Unified Modelling Language

12. REFERENSI

[GHJ+95] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, AND JOHN VLISSIDES. Design Patterns: Elements of Reusable Object-Oriented Software. 1995.

[KLM+97] GREGOR KICZALES, JOHN LAMPING, ANURAG MENHDHEKAR, CHRISMAEDA, JEAN-MARC LOINGTIER, AND JOHN IRWIN. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, ECOOP '97 Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, volume 1241, pages 220–242. Springer-Verlag, New York, NY, 1997.

[KHH+97] GREGOR KICZALES, ERIK HILSDALE, JIM HUGUNIN, MIK KERSTEN, JEFFREY PALM, AND WILLIAM GRISWOLD. Getting started with AspectJ. Communications of the ACM, 44(10):59–65, 2001.

[Lasiecki+02] NICHOLAS LESIECKI. Improve Modularity with Aspect-Oriented Programming. http://www-106.ibm.com/developerworks/java/library/j-aspectj/?dwzone=java. 2002.

[Laddad+02] RAMVINAS LADDAD, I want my AOP 1-3.

http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2-p3.html 2002

[Laddad+03] RAMVINAS LADDAD, Aspect Refactoring 1-2. http://www.theserverside.com/resources/article.jsp?l=AspectOrientedRefactoringPart. 2003

[Laddad+03] RAMVINAS LADDAD, AspectJ in Action. 2003

BIOGRAFI PENULIS

Adhari C Mahendra. Menyelesaikan program S1 Fakultas Ilmu Komputer di Universitas Indonesia tahun 2001. Saat ini bekerja di Jatis Solutions atau juga dikenal Firium di Singapura, Malaysia, dan Thailand, di Divisi Riset dan Pengembangan (aka TIG). Kompetensi inti adalah pada bidang Real Time Embedded System, Design Pattern, Software Engineering using Unified Process, Object-Orientation, Aspect-Orientation, Network and Operating System, Database, Distributed and Federated System, Parallel dan Grid Computing, Business Process Management, dan Digital and Automation Control.

Sekarang dibantu oleh Gregor Kickzales aktif merintis forum diskusi tentang AOP untuk wilayah Asia Tenggara. Anggota dari Sun Development Centre (SDC), Eclipse Development Centre, dan AOSD.