PENERAPAN *OBJECT RELATIONAL MAPPING* PADA *ENTERPRISE RESOURCE PLANNING*

Dian Indah Savitri[1)], **Wisnu Ananta Kusuma** [2)]
Departemen Ilmu Komputer, Fakultas Matematika dan Ilmu Pengetahuan Alam
Institut Pertanian Bogor (IPB)
Jl. Meranti Kampus IPB Darmaga-Bogor Telp/Fax : (0251) 625584
E-mail: [1)]divitri@gmail.com [2)] ananta@ipb.ac.id

Abstrak

ERP (Enterprise Resource Planning) merupakan suatu aplikasi integrasi yang difokuskan untuk mengotomasi seluruh aktivitas infrastruktur dalam suatu perusahaan. Sistem ERP menggabungkan proses bisnis antara perusahaan dan pelanggan, perusahaan dengan supplier, dan proses perhitungan finansial perusahaan. Semua proses bisnis yang tergabung mengakses pada sebuah basis data yang terpusat.

Sistem manajemen basis data yang banyak digunakan pada aplikasi ERP (Enterprise Resource Planning) adalah basis data relasional, sedangkan pengembangan aplikasi skala enterprise sebagian besar menerapkan konsep berorientasi objek. Dengan demikian terdapat ketidaksesuaian antara basisdata relasional dengan aplikasi yang menggunakan konsep berorientasi objek. Ketidaksesuaian tersebut antara lain terkait dengan aspek granularity, subtypes, identity, association, dan navigasi data.

Pada penelitian ini dilakukan analisis terhadap rancangan aplikasi Retail ERP. Ditemukan ketidak sesusian yang meliputi tiga aspek, yaitu aspek identitas, asosiasi, dan navigasi data. Untuk mengatasi ketidaksesuaian tersebut, diimplementasikan konsep Object Relational Mapping (ORM). Setiap objek yang akan dipetakan menjadi tabel-tabel pada basis data relasional dibungkus oleh suatu interface dengan menerapkan konsep design pattern. Hal tersebut bertujuan untuk memudahkan integrasi dengan lapisan aplikasi.

Implementasi ORM ini dibagi menjadi beberapa tahapan, yaitu: pemetaan Data Transfer Object dengan menghilangkan paradigma ketidaksesuaian antara basis data relasional dengan pemrograman berorientasi objek, membuat fungsi-fungsi akses data dengan mengimplementasikan DAO pattern, menyederhanakan fungsi akses data dengan memanfaatken konsep design pattern, membuat skema basis data, dan menyatukan lapisan persistensi dan lapisan aplikasi.

Implementasi konsep ORM terbukti dapat menghilangkan ketidaksesuian tersebut. Selain itu penerapan ORM yang dilengkapi dengan design pattern membuat sistem ERP menjadi lebih mudah untuk dikembangkan.

Keyword *: Object Relational Mapping (ORM), Enterprese Resource Planning (ERP), basis data, Object Oriented, Design Pattern.*

1. PENDAHULUAN

ERP (*Enterprise Resource Planning*) merupakan suatu aplikasi terintegrasi yang difokuskan untuk mengotomasi seluruh aktivitas infrastruktur dalam suatu perusahaan. Sistem ERP menggabungkan proses bisnis antara perusahaan dan pelanggan, perusahaan dengan *supplier*, dan proses perhitungan keuangan perusahaan. Semua proses bisnis yang tergabung mengakses pada sebuah basis data yang terpusat (Parr 2000)..

Aplikasi yang berskala *enterprise* dikembangkan dengan pendekatan berorientasi objek menggunakan *three-tier-architecture* yaitu *presentation layer*, *lapisan aplikasi*, dan *database layer* (Rashid, *et. al* 2002). Pada penelitian ini, arsitektur tersebut diakomodasi oleh Java EE dengan konsep MVC (*Model*, *View*, dan *Controller*). Konsep MVC memisahkan kode untuk tampilan dan proses bisnisnya. Bagian proses bisnis berisi tugas-tugas validasi, *workflow*, dan manajemen basis data.

Sementara itu, sistem manajemen basis data yang banyak digunakan sekarang ini adalah basis data relasional. Sedangkan pengembangan aplikasi skala *enterprise* sebagian besar menerapkan konsep berorientasi objek (Mehra K K, *et al.* 2007). Dengan demikian, terdapat ketidaksesuaian antara basisdata relasional dengan pengembangan aplikasi yang menggunakan konsep berorientasi objek. Ketidaksesuaian tersebut antara lain berkaitan dengan aspek *granularity*, *subtypes*, identitas, asosiasi dan navigasi data (Bauer dan King 2007).

Object Relational Mapping (ORM) adalah sebuah *framework* yang dapat menjembatani perbedaan sistem basis data yang bersifat relational dengan paradigma pengembangan aplikasi yang berorientasi objek. Setiap objek yang akan memetakan menjadi tabel-tabel pada basis data relasional dibungkus oleh suatu *interface* dengan menerapkan konsep *design pattern*. Hal tersebut bertujuan untuk memudahkan lapisan aplikasi (*controller*) mengakses data tersebut.

Tujuan dari penelitian ini yaitu menganalisis ketidaksesuaian (*mismatch*) yang muncul dari rancangan Retail ERP yang dikembangkan Ernita (2008) dan basis data relasional yang digunakan. Selanjutnya akan diterapkan konsep *Object Relational Mapping* untuk mengatasi masalah ketidaksesuaian tersebut. Penelitian ini juga akan memanfaatkan *design pattern* dalam pengelolaan akses basis data oleh lapisan aplikasi dan presentasi dengan harapan dapat bermanfaat dalam pemeliharaan dan pengelolaan manajemen basis data pada pengembangan sistem ERP selanjutnya.

2. TINJAUAN PUSTAKA

ERP (*Enterprise Resource Planning*)

ERP (*Enterprise Resource Planning*) adalah suatu aplikasi terintegrasi yang berkonsentrasi untuk menyatukan aktivitas proses transaksi *enterprise*. Aplikasi ERP menggabungkan proses bisnis antara perusahaan dan pelanggan, perusahaan dengan *supplier*, dan proses perhitungan finansial perusahaan. ERP mengautomatisasi proses bisnis perusahaan baik dari segi produksi, distribusi, finansial, waktu, dan sumber daya manusia (Parr 2000). ERP menyediakan solusi dari berbagai macam permasalahan pada industri dan infrastruktur yang terintegrasi (Themistocleous 2001).

Object persistence

Dalam pengembangan sistem, pengertian *persistence* adalah objek yang dihasilkan dari suatu sistem yang dapat disimpan dalam waktu lama bahkan bersifat permanen (Peak dan Heudecker 2006). *Object Persistence* bisa disimpan dalam bentuk basis data relasional, *file system* dalam *harddisk*, *file* XML, *Web Service*, *ODBMS (Object Data Base Management System),* dan LDAP. Namun *Object Persistence* yang paling banyak digunakan dalam pengembangan perangkat lunak adalah basis data relasional. Basis data relasional mudah dibuat dan diakses.

Dengan *object persistence*, data yang terkandung pada objek tersebut dapat dengan mudah disimpan dan diakses. *Object persistence* bersifat abstrak, dapat menyembunyikan rincian mekanisme bagaimana suatu data disimpan dan diakses, seingga tidak terlihat oleh objek lainnya.

Object-relational mismatch

Object-relational mismatch adalah ketidaksesuaian antara basis data yang menggunakan konsep relasional dengan pengembangan aplikasi yang menggunakan konsep berorientasi objek. Ketidaksesuaian tersebut meliputi aspek:

1. *Granularity*
   Pemecahan *entity* pada atribut-atribut yang lebih kompleks. Misalnya *class* Person mempunyai atribut address dengan tipe data Address. Sedangkan pada basis data relasional tidak memungkinkan pada tabel PERSON ada kolom address dengan tipe data address, yang mungkin dilakukan adalah memecah address menjadi street_address, city_address, zipCode_address.
2. *Subtypes*
   Pembeda antara *superclass* dan *subclass*. Pada pemrograman berbasis objek dikenal istilah *inheritance* (pewarisan) dari *class parent* kepada *class child*. Sedangkan pada basis data relasional tidak dikenal proses pewarisan antar tabel.
3. *Identity*
   Terdapat perbedaan fungsi antara lambang sama dengan (=) dan *method equals(),* pada objek tetapi merujuk nilai yang sama pada *primary key* basis data relasional.
4. *Association*
   Hubungan dua entitas pada objek dikenal dengan *references* sedangkan pada relasional dikenal dengan *foreign key*. Asosiasi antar entity terdiri dari: *one-to-one*, *one-to-many*, dan *many-to-many*.
5. Navigasi data
   Proses pencarian pada basis data menggunakan *join table* sedangkan pada objek memanggil suatu *method getter*. Navigasi dibagi dua macam berdasarkan arahnya, antara lain: *directional* dan *bidirectional*. (Bauer dan King 2007)

Object Relational Mapping

Object Relational Mapping merupakan teknik otomasi dan transparansi dari *object persistence* ke dalam tabel pada basis data, menggunakan metadata yang mendeskripsikan pemetaan antara objek dan basis data (Bauer dan King 2007). ORM berperan dalam lapisan model dalam konsep MVC. Model adalah sebuah lapisan yang paling dekat dengan sumber data, baik itu berasal dari basis data, *webservice*, maupun *file system*.

Object Relational Mapping (ORM) adalah sebuah *framework* yang mengatasi perbedaan sistem basis data yang bersifat relational dengan paradigma pengembangan aplikasi yang berorientasi objek. Selain itu, ORM juga menjembatani dialek SQL yang digunakan, sehingga apapun produk RDBMS yang digunakan tidak berpengaruh terhadap kode program. ORM merupakan solusi yang mengatasi perbedaan aspek-aspek ketidaksesuaian antara konsep pemrograman berorientasi objek dengan konsep basis data relasional.

Framework

Framework adalah aplikasi *semi-complete* yang dapat digunakan untuk membuat aplikasi lain yang lebih spesifik (Husted 2003). *Framework* yang ideal merupakan intisari dari pendekatan terbaik dalam pengembangan perangkat lunak. Alasan pengembangan sebuah *framework* adalah penggunaan kembali perangkat lunak, jadi suatu perangkat lunak dikembangkan dari perangkat lunak yang telah ada.

Framework yang mempunyai konsep ORM salah satunya adalah Hibernate. Hibernate dikembangkan untuk menangani masalah *object persistence* pada lingkungan Java. Implementasi Hibernate pada kode program Java menggunakan *Hibernate Query Language* (HQL) (Bauer dan King 2007).

Design Pattern

Design pattern adalah unsur-unsur rancangan yang seringkali muncul pada berbagai jenis sistem. *Design pattern* merupakan katalog permasalahan beserta solusi yang sudah teruji dalam perancangan sistem (Gamma *et al*. 1995). *Design pattern* yang sering dijadikan untuk bidang studi adalah *design pattern* yang berasal dari *Gang of Four* (GoF) yaitu, Erich Gamma, Richard Helm, Ralph Johnson, dan John Vlissides.

Design pattern yang digunakan pada penelitian ini adalah *singleton, proxy*, DAO (Data *Access Object*), *façade*, dan *factory method*. *Singleton* adalah pola yang hanya mempunyai satu instansiasi dari suatu *class* yang dipakai pada sistem tertentu. *Proxy* adalah pola yang merancang suatu objek mewakili kontrol atau akses objek lain bertindak seolah-oleh objek tersebut melakukannya. *Façade* adalah pola yang menyederhanakan objek-objek yang terlihat rumit ke dalam sebuah *interface*. *Factory method* adalah pola untuk mengenkapsulasi suatu objek dan diinstasiasi satu kali untuk digunakan berulang-ulang. Sedangkan *Data Access Object* (DAO) *pattern* adalah pola yang mengenkapsulasi data akses dan manipulasi ke dalam lapisan yang berbeda kemudian diimplementasikan dengan membuat sebuah objek yang bersifat abstrak dan mengenkapsulasi seluruh akses data. (Alur *et al*. 2003).

Declarative Transaction

Declarative transaction adalah suatu teknik untuk medeklarasikan semua transaksi dalam suatu *file* konfigurasi untuk menangani semua proses transaksi tanpa harus membuat kode-kode program untuk menangani transaksi secara eksplisit. Teknik ini menggunakan metode pemrograman AOP (*Aspect Oriented Programming*). AOP mendefinisikan method yang akan dipakai untuk transaksi dengan format ekspresi regular. Selain itu mengatur *rollback* (*method* dan *exception*) apabila terjadi kegagalan transaksi (Walls dan Beidenbach 2005).

Dependency Injection / Inversion of Control

Dependency Injection adalah suatu konsep yang diterapkan oleh suatu objek untuk mendapatkan objek lain yang dibutuhkan tanpa harus mencari dan memasukan objek tersebut secara eksplisit (Walls dan Beidenbach 2003). Konsep ini diterapkan dalam Spring *framework*, dimana untuk semua DAO yang didefinisikan tidak harus menuliskan koneksi basis datanya secara langsung pada masing-masing DAO, tetapi cukup memasukkan koneksinya menjadi suatu *construktor* atau *method setter* dalam DAO (Thamura 2007).

3. METODE PENELITIAN

Penelitian ini diawali dengan studi literatur untuk mendapatkan informasi terkait dengan konsep ORM dan ERP. Selanjutnya dilakukan analisis untuk mengidentifikasi aspek-aspek ketidaksesuaian (*mismatch*) antara aplikasi ERP yang dikembangkan berdasarkan pendekatan berorientasi objek dan basis data relasional yang digunakan. Tahap selanjutnya dilakukan implementasi ORM untuk mengatasi ketidaksesuaian tersebut. Tahap implementasi ORM ini dibagi menjadi lima, yaitu: pemetaan *Data Transfer Object* dengan menghilangkan paradigma ketidaksesuaian antara basis data relasional dengan pemrograman berorientasi objek, membuat fungsi-fungsi akses data dengan mengimplementasikan DAO *pattern*, menyederhanakan fungsi akses data dengan memanfaatken konsep *design pattern*, membuat skema basis data, dan menyatukan lapisan persistensi dan lapisan aplikasi. Tahap terakhir penelitian ini adalah melakukan evaluasi hasil implementasi ORM pada ERP yang terkait dengan aspek penghematan sumber daya untuk koneksi pada basis data dan kemudahan dalam pengelolaan konfigurasi dan pengembangan aplikasi selanjutnya.

4. HASIL DAN PEMBAHASAN

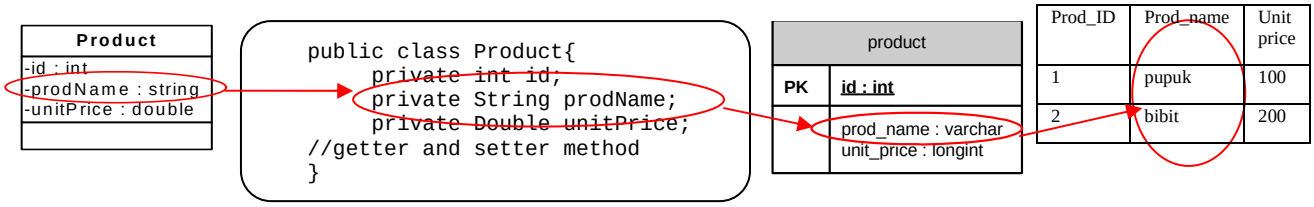
4.1 Masalah Ketidaksesuaian pada Aspek Identitas, Asosiasi, dan Navigasi

Penelitian ini memanfaatkan aplikasi Retail ERP yang dikembangkan Ernita (2007). Sistem ERP ini membutuhkan basis data dalam menjalankan proses bisnisnya. Tiga modul utama pembangun sistem ERP mengakses pada satu basis data yang terpusat. Pada penelitian ini, sistem ERP dikembangkan dengan pendekatan berorientasi objek. Sedangkan basis data yang digunakan adalah basis data relasional. Oleh karena itu, dilakukan analisis terhadap ketidaksesuaian antara konsep basis data relasional dengan konsep orientasi objek. Pada penelitian ini, tidak semua aspek ketidaksesuaian ditemukan. Aspek ketidaksesuaian yang muncul pada penelitian ini adalah aspek identitas, navigasi data, dan asosiasi antar entitas.

Ketidaksesuaian pada aspek identitas terjadi pada semua *class Data Transfer Obejct* (DTO), karena setiap class mempunyai pembeda dalam mengenali kesamaan identitas yaitu secara lojik `(a.equals(b))` dan secara lokasi memori memegang alamat sama, namun belum tentu mempunyai nilai yang sama `(a == b)`. Sementara itu ketidaksesuaian pada aspek asosiasi terjadi pada keterhubungan one-to-one, one-to-many, dan many-to-many. Sedangkan ketidaksesuain yang terjadi pada aspek navigasi data disebabkan perbedaan mekanisme mengakses data pada objek dan tabel.

4.2 Impelementasi ORM untuk Mengatasi Masalah Ketidaksesuaian

DTO merupakan *class* yang merepresentasikan setiap tabel pada basis data. DTO berfungsi sebagai jembatan penghubung antar lapisan dalam aplikasi. Dengan pola ini proses perpindahan data menjadi sederhana dan terintegrasi. DTO dibuat berdasarkan UML *class* diagram yang telah dirancang. DTO berisi *class JavaBean* yang setiap propertinya akan merepresentasikan atribut-atribut pada tabel. Skema proses pembuatan DTO diilustrasikan pada Gambar 1 berikut :



Gambar 1 proses pemetaan *class* DTO menjadi tabel pada basis data.

Proses pemetaan *class JavaBean* (DTO) menjadi tabel terjadi ketidaksesuaian antara tabel yang berasal dari basis data relasional dan *class javaBeans* yang berorientasi objek. Ketidaksesuaian tersebut dapat diatasi dengan menggunakan konsep ORM sehingga setiap tabel bisa merepresentasikan *class* DTO dan diakses sebagai objek. Uraian berikut menjelaskan implementasi konsep ORM untuk mengatasi ketidaksesuaian yang meliputi tiga aspek yaitu aspek identitas, asosiasi, dan navigasi.

a. Identitas

Terdapat perbedaan pengenal identitas antara objek dan tabel. Pada objek identitas dibedakan menjadi nilai dan alamat memorinya. Sehingga terdapat dua notasi yang melambangkan kesamaan suatu identitas pada objek Java, yaitu lambang samadengan `(==)` dan *method* `equals()`. Contoh : `a == b`, berarti variabel a dan b memegang alamat *reference* yang sama pada memori. `(a.equals(b))`, secara lojik mempunyai nilai yang sama.

Pada basis data relasional, identitas dari suatu tabel disebut dengan *primary key*. Dengan demikian, sering kali terjadi objek yang secara lojik sama (`a.equals(b)`) dan mewakili satu baris dalam tabel basis data, tetapi objek tersebut tidak berada pada satu lokasi alamat memori yang sama.

ORM mengatasi ketidaksesuaian tersebut dengan menambah properti *identity* pada setiap entity atau *class* DTO seperti pada potongan program berikut:

```
1 @Entity
2 @Table(name = "IN_ITEM")
3 public class InventoryItem{
4   @Id
6   @Column(name="id")
7   private Long id;
8   //definisi kolom-kolom
9  //getter dan setter method
   }
```

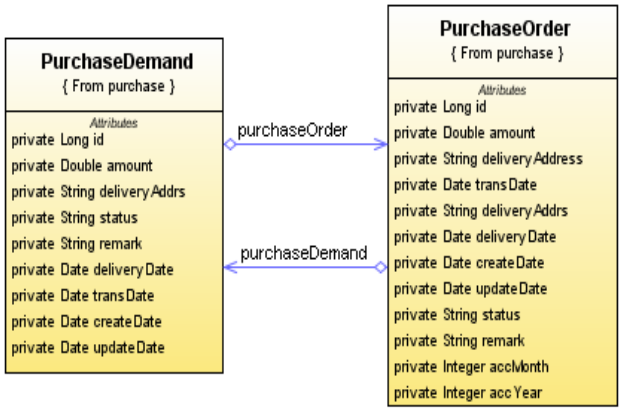
Pada baris pertama dan kedua menandakan bahwa *class* tersebut mewakili sebuah entitas pada basis data dengan nama tabel IN_ITEM. `@Id` pada baris keempat merupakan properti yang merepresentasikan *primary key*, secara lojik properti `@Id` pada *class* a berbeda dengan `@Id` pada *class* b. Dengan demikian pengujian apakah *class* a sama dengan *class* b bisa ditulis seperti berikut: `a.getid().equals(b.getId())`.

b. Asosiasi

Kebanyakan entitas pada basis data mempunyai keterhubungan dengan entitas yang lainnya. Elemen yang menghubungkan kedua tabel tersebut ditandai dengan *foreign key*. Pada objek, elemen penghubung dua objek ditandai dengan sebuah *reference object*. Namun permasalannya adalah pada *reference object* yang menghubungkannya bergantung pada arah asosiasinya (*direction of relationship*). Apabila asosiasi antara dua objek terjadi pada dua arah, maka harus didefinisikan dua kali pada setiap *class*nya.

Akan tetapi *foreign key* pada tabel relasional, tidak mengenal arah dari asosiasinya, karena asosiasi antara dua tabel dihubungkan dengan *table join* atau *projection.* Asosiasi antara dua entitas terdiri dari *one-to-one*, *one-to-many*, dan *many-to-many*. Proses pemetaan asosiasi antara dua entitas adalah sebagai berikut:

- **One-to-one**



Gambar 2 contoh pemetaan asosiasi *one-to-one.*

Sebagai contoh pada Gambar 2 *class* PurchaseDemand beasosiasi *one-to-one* dengan *class* PurchaseOrder. Letak *foreign key* atau *join column* ditambahkan pada entitas yang mempunyai *total participant* pada asosiasi yang menghubungkannya. Pada entitas yang mempunyai asosiasi dua arah (*bidirectional*) adanya penambahan properti `mappedBy` oleh entitas yang bukan *total participant* (entitas *partial participant*). Sedangkan pada entitas yang mempunyai asosiasi satu arah (*unidirectional*) tidak perlu menambahkan properti `mappedBy`.

Proses pemetaan dua *class* PurchaseDemand dan *class* PurchaseOrder merupakan contoh asosiasi dua arah (*association bidirectional*) dan *class* PurchaseDemand mempunyai *total participant.* Properti `Join Column` dimasukan pada *class* PurchaseDemand karena *class* PurchaseDemand mempunyai *total participant*, artinya pada contoh asosiasi *one-to-one* di atas, objek PurchaseOrder bisa berdiri sendiri tanpa objek PurchaseDemand, sedangkan objek PurchaseDemand ridak dapat berdiri sendiri tanpa objek PurchaseOrder. Sehingga *class* PurchaseDemand harus berpartisipasi secara *total* pada contoh asosiasi *one-to-one* tersebut.

Asosiasi *one-to-one* di atas bersifat *biderectional* karena masing-masing *class* memerlukan *reference* yang menghubungkannya. Pada *class* PurchaseDemand *reference* ditulis seperti pada potongan program berikut:

```
@OneToOne
@JoinColumn (name="id_pu_order")
private PurchaseOrder order;
```

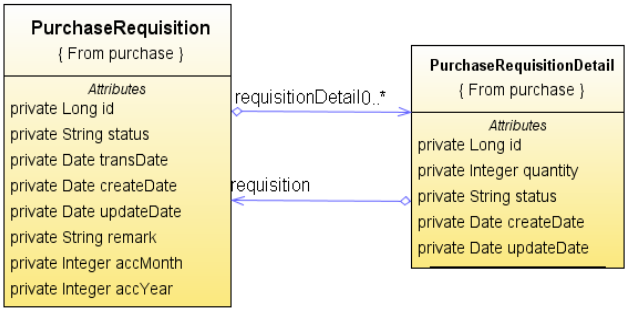
dan pada *class* PurchaseOrder, *reference* yang menghubungkannya ditulis dengan `mappedBy` seperti potongan program berikut:

```
@OneToOne (mappedBy="order")
private PurchaseDemand purDemand;
```

properti `mappedBy="order"` pada class PurchaseOrder merujuk pada nama atribut `order` pada *class* PurchaseDemand.

- **One-to-many**

Sebagai contoh *class* PurchaseRequisition beasosiasi *one-to-many* dengan *class* PurchaseRequisitionDetail seperti diilustrasikan pada Gambar 3. Jadi setiap satu objek PurchaseRequisition terdiri dari banyak objek PurchaseRequisitionDetail. Sama halnya seperti asisasi *one-to-one*, pada asisasi *one-to-many* letak *join column* berada pada class yang mempunyai *total participant*. Akan tetapi pada *association one-to-many*, *class* yang mempunyai total *participant* pasti berada pada class yang mempunyai kardinalitas *many*. Dengan demikian, dapat dikatakan *join column* berada pada *class* dengan kardinalitas *many*.



Gambar 3 contoh pemetaan asosiasi *one-to-many*.

Class PurchaseRequisition beasosiasi *one-to-many* dengan *class* PurchaseRequisitionDetail. Apabila asosiasinya *bidirectional*, maka dapat dikatakan *class* PurchaseRequisitionDetail beasosiasi *many-to-one* dengan *class*. Contoh pemetaan asosiasi *many-to-one* di atasi bersifat biderectional, sehingga `join column` berada pada *class* yang mempunyai kardinalitas *many* yaitu *class* PurchaseRequisitionDetail sedangkan pada *class* dengan kardinalitas *one* ditambahkan properti `mappedBy` sebagai *reference* dari *class* PurchaseRequistionDetail.

Pada *class* PurchaseRequisitionDetail reference ditulis seperti pada potongan berikut:

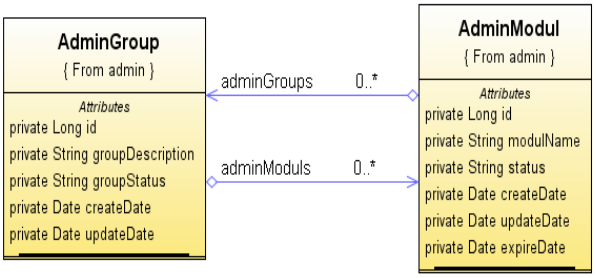
```
@ManyToOne
@JoinColumn ( name = " id_pu_requisition")
private PurchaseRequisition requisition;
```

dan pada *class* PurchaseRequisition, reference yang menghubungkannya ditulis dengan *mappedBy* seperti potongan program berikut:

```
@OneToMany (mappedBy = "requisition")
private List <PurchaseRequisitionDetail> requisitionDetail = new ArrayList
<PurchaseRequisitionDetail>();
```

Apabila asosiasinya *unidirectional*, properti `mappedBy` pada *class* PurchaseRequisition dihilangkan. Properti `mappedBy = "requisition"` merujuk kepada atribut `"requisition"` pada *class* PurchaseRequisitionDetail. Proses pemetaan *class* PurchaseRequisition yang berasosiasi *one-to-many* dengan *class* PurchaseRequisitionDetail dapat dikatakan juga sebagai proses pemetaan *parent child*. Entitas yang mempunyai kardinalitas *one* akan menentukan entitas yang mempunyai kardinalitas *many*. Apabila terjadi proses *save*, *update*, *delete* pada entitas yang berkardinalitas *one*, maka entitas yang berkardinalitas *many* akan terpengaruh. Dan sebaliknya, entitas yang berkardinalitas *one* tidak bisa berdiri sendiri tanpa adanya entitas yang berkardinalitas *many*.

- **Many to many**



Gambar 4 contoh pemetaan asosiasi *many-to-many*.

Sebagai contoh pada Gambar 4 *class* AdminGroup berasosiasi *many-to-many* dengan *class* AdminModul. Atribut *reference* yang menghubungkan kedua class tersebut adalah join tabel. Join tabel memiliki dua buah atribut *foreign key* yang berasal dari masing-masing tabel.

Atribut `join tabel` berada pada class yang memilki *total participant* pada asosiasi tersebut. Pada entitas yang mempunyai asosiasi dua arah (*bidirectional*) adanya penambahan properti `mappedBy` oleh entitas yang bukan *total participant* (entitas *partial participant*). Sedangkan pada entitas yang mempunyai asosiasi satu arah (*unidirectional*) tidak perlu menambahkan properti `mappedBy`.

Proses pemetaan dua *class* AdminModul dan *class* AdminGroup merupakan contoh asosiasi dua arah (*association bidirectional*) dan *class* AdminModul mempunyai *total participant*. Properti `Join table` dimasukan pada properti *class* AdminModul karena *class* AdminModul mempunyai *total participant*, artinya pada *association many-to-many* di atas, objek AdminGroup bisa berdiri sendiri tanpa objek AdminModul, sedangkan objek AdminModul ridak dapat berdiri sendiri tanpa objek AdminGroup.

Asosiasi *many-to-many* disebut *biderectional* karena masing-masing *class* memerlukan *reference* yang menghubungkannya. Pada class AdminModul *reference* ditulis seperti pada potongan program berikut:

```
@ManyToMany
@JoinTable(name=  "ad_modul_group",         joinColumns={@JoinColumn(name=  "id_ad  _modul")},
inverseJoinColumns={@JoinColumn(name="id_ad_group")})
private List <AdminGroup>  adminGroups=new ArrayList <Admin Group>();
```

dan pada *class* AdminGroup, reference yang menghubungkannya ditulis dengan *mappedBy* seperti potongan program berikut:

```
@ManyToMany(mappedBy="adminGroups")
private List <AdminModul> admin Moduls = new ArrayList<AdminModul>();
```

Properti `mappedBy="adminGroups"` pada class AdminGroup merujuk pada nama atribut `adminGroups` pada *class* AdminModul. Hasil pemetaan dari asosiasi many-to-many dua class tersebut adalah dua tabel hasil transformasi dari objek yaitu AD_MODUL dan AD_GROUP kemudian satu tabel asosiasi AD_MODUL_GROUP.

c. Navigasi data

Masalah navigasi data adalah problem bagaimana mengakses suatu objek dari objek lain. Menurut arahnya, navigasi terbagi menjadi dua macam, yaitu: *unidirectional* dan *bidirectional*. Pada premrograman berbasis objek, proses akses properti objek dari objek lain bisa langsung menggunakan *method getter*. Contoh:

```
    // Class InvItem
@Entity
@Table(name = "IN_ITEM")
public class InvItem {
    @Id
    private Long id;
    @Colums(name="in_desc")
    Private String inDesc;
    @ManyToOne
    @JoinColumn(name="id_in _brand")
    private InvProdBrand invProdBrand;
//getter dan setter method
}
```

Untuk mengakses objek invProdBrand dengan tipe data InvProdBrand menggunakan *method getter* seperti berikut: `invItem.getInvProdBrand.getBrandDescription();`. Apabila arah navigasinya *unidirectional*, maka tidak terdapat properti objek invItem pada class InvProdBrand, sehingga *class* invProdBrand tidak bisa mengakses properti objek invItem. Tetapi apabila arah navigasinya *bidirectional*, maka pada *class* invProdBrand terdapat properti objek invItem seperti berikut:

```
    //class InvProdBrand
@Entity
@Table(name="IN_PROD_BRAND")
public class InvProdBrand {
    @Id
    private Long id;
    @Column(name="brand_desc")
    private String brandDesc;
    @OneToMany(mappedBy="inv ProdBrand")
    private List <InvItem> invItem = new ArrayList<InventoryItem>();
```

properti `mappedBy` menandakan bahwa *class* tersebut mempunyai asosiasi *bidirectional*. Untuk mengakses properti objek invItem dari *class* InvProdBrand menggunakan *method getter* seperti berikut: `invProdBrand.getInItem.getInDesc();`.

Sedangkan pada basis data relasional, konsep arah navigasi tidak mempengaruhi proses akses data. Tabel IN_ITEM dan IN_PROD_CAT mempunyai relasi *one-to-many* kemudian *foreign key* berada pada tabel IN_ITEM, akan tetapi arah navigasi dari asosiasi yang menghubungkannya tidak berpengaruh. Selama ada *foreign key* yang menghubungkan kedua tabel, proses query yang melibatkan kedua tabel bisa dijalankan.

4.3 Fungsi Akses Data dan Pemanfatan Design Pattern

Pada penelitian ini, untuk mengenkapsulasi data akses dan manipulasi ke dalam lapisan yang berbeda digunakan DAO pattern. DAO ini diimplementasikan menggunakan konsep ORM dengan menggunakan HQL (*Hibernate Query Language*). Proses pembuatan akses dan manipulasi data menggunakan DAO *pattern* juga memanfaatkan konsep *design pattern factory method* dan *singleton*.

Sebelum melakukan akses dan manipulasi data, maka dibuat objek `dataSource` yang berisi referensi dari properti koneksi basis data. Objek `dataSource` dan semua objek hanya diinstansiasi satu kali selama aplikasi berjalan. Objek `dataSource` dipanggil setiap DAO dijalankan, sehingga proses koneksi basis data tidak dilakukan secara berulang-ulang melainkan cukup dipanggil apabila diperlukan.

Pada penelitian ini, proses pembuatan `dataSource` dan pemanggilan `dataSource` oleh semua DAO dilakukan menggunakan Spring Framework dengan teknik *Inversion of Control* (IoC). Tehnik IoC untuk `dataSource` ditulis pada file konfigurasi `hibernate`.ctx.xml seperti berikut:

```
<bean id = "dataSource" class="org.springframework.jdbc. datasource.DriverManagerDataSource">
    <property name = "username" value = "${jdbc.username}"/>
    <property name = "password" value = "${jdbc.password}"/>
    <property name = "url" value = "${jdbc.url}"/>
    <property name = "driverClassName" value = "${jdbc.driver}"/
</bean>
```

Framework ORM yang digunakan pada penelitian ini adalah Hibernate. Method-method yang terdapat pada DAO diimplementasikan dengan HQL(*Hibernate Query Language*). DAO berisi *method* untuk akses dan manipulasi data seperti *create, update, delete, insert, select,* dan *join.* Sebelum melakukan transaksi untuk akses dan manipulasi data, dibuat objek `SessionFactory` yang dihasilkan oleh konfigurasi properti Hibernate yang akan digunakan. Konfigurasi untuk Hibernate merujuk pada file konfigurasi untuk basis data `jdbc.properties`.

```
1<property name = "hibernateProperties">
2   <props>
3      <prop key="hibernate.show_ sql">${hibernate.show_sql}</prop>
4      <prop key="hibernate.format_ sql">${hibernate.format_sql}</prop>
5      <prop key="hibernate. dialect">${hibernate.dialect}</prop>
6      <prop key="hibernate.hbm2ddl. auto">${hibernate.hbm2ddl.auto}</prop>
7   </props>
8</property>
```

Konfigurasi yang telah diaktifkan akan menghasilkan objek `sessionFacory`. Seperti halnya `dataSource`, objek `sessionFactory` bersifat *singleton*, jadi objeknya hanya dibuat sekali sepanjang aplikasi berjalan. `sessionFactory` dipanggil oleh setiap class DAO untuk melakukan akses dan manipulasi basis data.

Objek `sessionFactory` mencari file konfigurasi `hibernate.ctx.xml`. Untuk melakukan akses dan manipulasi data, `sessionFactory` membuat objek `session. Session` merupakan objek yang hanya sekali pakai, digunakan untuk melakukan transaksi basis data dan setelah selesai transaksi, `session` langsung ditutup. Siklus hidup objek `session` berbeda dengan objek `sessionFactory`, `sessionFactory` dibuat sekali sepanjang aplikasi berjalan, sedangkan `session` dibuat selama proses suatu transaksi berlangsung. `Session` bertanggungjawab selama proses transaksi, `session` tidak bisa ditutup sebelum transaksi berhasil dan `session` akan melakukan *rollback* apabila transaksi gagal.

4.4 Penyederhanaan Fungsi Akses Data

Pada penelitian ini terdapat 50 class DTO yang masing-masing mempunyai *interface* DAO untuk akses dan manipulasi data seperti *insert, select,* atau *delete*. Untuk memudahkan dan akses *interface* DAO-DAO dikumpulkan berdasarkan submodul ERP ke dalam *interface* `Service`. Interface `Service` merupakan implementasi dari *facade pattern*, yaitu menyederhanakan fungsi-fungsi yang terlihat rumit dengan mengelompokan fungsi-fungsi tersebut ke dalam beberapa *package* sehingga menghasilkan fungsi yang lebih sederhana dan mudah digunakan.

Setelah disederhanakan dari 50 *class* DTO dihasilkan tujuh interface `Service` sesuai dengan submodulnya. Proses memasukan DAO kepada Service menggunakan tehnik *Inversion of Control* oleh Spring *Framework*. Seperti halnya DAO, Service merupakan suatu interface sedangkan implementasi dari interface tersebut berisi semua DAO untuk melakukan akses dan manipulasi basis data.

`Service` adalah properti yang akan diakses oleh application dan presentastion layer dalam proses akses basis data. Sebelum dapat diakses, dilakukan proses *Declarative Transaction* terlebih dahulu terhadap objek `Service` dengan menerapkan konsep *proxy pattern*. *Proxy pattern* yaitu pola yang merancang suatu objek untuk mewakili kontrol atau akses objek lain bertindak seolah-oleh objek tersebut melakukannya. Proses *Declarative Transaction* ditulis pada file `hibernate.ctx.xml`. Contoh penulisan *Declarative Transaction* pada modul Inventory dapat dilihat seperti berikut:

```
<beanid="moduleInventoryService"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="target" ref= "inventoryService"/>
    <property name="transactionManager" ref = "hibernateTransactionManager"/>
    <property name="transaction Attributes">
      <props>
          <prop key="*"> PROPAGATION_REQUIRED, readOnly
          </prop>
          <prop key="save*"> PROPAGATION_REQUIRED
            </prop>
            <prop key="delete*"> PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>
```

Objek `session` yang digunakan saat transaksi dikendalikan oleh *class* `proxy` yang dihasilkan melalui tehnik *Declarative Transaction*. Ketika lapisan aplikasi memanggil `Service` untuk akses basis data, `Service` tidak mengakses basis data secara langsung. Akan tetapi `Service` memanggil `moduleXXXService` pada file `hibernate.ctx.xml` untuk membuat *class* `proxy` yang berisi fungsi-fungsi transaksi (*begin*, *commit*, *rollback*). Fungsi-fungsi transaksi tersebut dimasukan ke dalam `session` melalui *method* `getHibernateTemplate`, sehingga tidak perlu menuliskan proses transaksi secara programatik mengendalikan `session`, karena transaksi dilakukan secara deklarasi oleh *class* `proxy` melalui method `getHibernateTemplate`.

4.5 Penyatuan Lapisan Persistensi dan Lapisan Aplikasi.

Framework yang digunakan pada lapisan aplikasi dan presentasi adalah Java Server Faces (JSF). JSF adalah UI(*User Interface*) *framework* untuk aplikasi Java berbasis web dengan konsep MVC yang mendukung *user interface* berbasis komponen. Setiap *action* yang dilakukan memerlukan *class controller* atau *backing bean*. Pada *class controller* biasanya dilakukan akses dan manipulasi terhadap basis data.

Class controller yang akan melakukan akses atau manipulasi data memanggil *method* `Service` sesuai data yang diperlukan. Misalnya untuk akses *class* `InventoryItem`. DAO yang berisi *method-method* untuk akses *class* `InventoryItem` disimpan pada `InventoryItemDao`, kemudian `InventoryItemDao` dimasukan ke dalam `InventoryService`. Oleh karena itu, untuk mengakses `InventoryItem` dipanggil `InventoryService`. Dan terakhir mengatur konfigurasi *controller* untuk JSF pada *file* `faces-config.xml`. *method* `Service` yang digunakan oleh *controller* dideklarasikan pada *file* konfigurasi `faces-config.xml`.

4.6 Evaluasi

Aplikasi *enterprise* biasanya mengoptimasi koneksi basis data menggunakan konsep *connection-pooling*. Pada saat aplikasi mulai berjalan, aplikasi langsung membuat banyak koneksi sekaligus (*pool*). Apabila diperlukan akses basis data, maka koneksi diambil dari *pool*, setelah selesai mengakses basis data, koneksi tidak ditutup melainkan dikembalikan ke *pool*. Dengan ORM, semua koneksi terhadap basis data diatur pada satu tempat, sehingga apabila akan mengubah konfigurasi dari koneksi biasa menjadi *connection-pooling* tidak akan berpengaruh terhadap keseluruhan kode program.

Tidak semua data yang diambil dari basis data akan digunakan, kadang kala sebagian data diambil untuk dihapus. Proses tersebut dapat memboroskan kinerja CPU, penggunaan memori dan *bandwidth*. ORM mempunyai kelebihan yaitu mengambil data apabila benar-benar diperlukan.ORM memiliki kontrol penuh terhadap koneksi basis data, sehingga dapat dengan mudah mengamati data yang jarang berubah dan kemudian menyimpannya di memori. Hal ini akan mempercepat proses pengaksesan basis data, karena ORM tidak langsung mengakses basis data, melainkan langsung mengambilnya dari memori.

Pemanfaatan konsep *design pattern* dapat terlihat dengan kode program yang lebih rapi dan terstruktur. Serta memudahkan untuk proses integrasi antara lapisan basis data dengan lapisan aplikasi. Dengan penerapan konsep *design pattern*, kode program aplikasi dapat dengan mudah dipelihara untuk pengembangan aplikasi selanjutnya.

5. KESIMPULAN

Pada penelitian ini, tidak semua aspek ketidaksesuaian ditemukan. Aspek ketidaksesuaian yang muncul pada penelitian ini hanya meliputi tiga aspek, yaitu aspek identitas, navigasi data, dan asosiasi antar entitas. Implementasi konsep ORM terbukti dapat menghilangkan ketidaksesuian yang muncul karena perbedaan antara aplikasi yang dibangun dengan pendekatan berorientasi objek dan basis data relasional yang digunakan. Selain itu penerapan ORM yang dilengkapi dengan *design pattern* membuat sistem ERP menjadi lebih terstruktur khususnya dalam pengelolaan basis data. Penerapan *design pattern* pada teknologi *declarative transaction* dan *Inversion of Control* dapat memudahkan proses akses dan manipulasi pada lapisan aplikasi. Dengan demikian memudahkan untuk melakukan pengembangan aplikasi ERP selanjutnya.

6. DAFTAR PUSTAKA

Alur D, Crupi J, Malks D. 2003. *Core J2EE™ Patterns: Best Practices and Design Strategies, 2nd Edition.* United States: Addison-Wesley.

Bauer C, King G. 2007. *Java Persistence with Hibernate.* United States : Manning.

Ernita, H. 2008. Pengembangan Enterprise Resource Planning Untuk Perusahaan Ritel. [*Skripsi*]. Bogor: Departemen Ilmu Komputer, Institut Pertanian Bogor.

Gamma E, Helm R, Johnson R, Vlissides J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. United States: Addison-Wesley.

Husted T. 2003. *Struts in Action : Building Web Application with the Leading Java Framework.* United States : Manning.

Parr A. N. 2000. *A Taxonomi of ERP Implementation Approach*. School of Business System, Monash University.

Peak P, Heudecker N. 2006. *Hibernate Quickly.* United States: Manning.

Thamura F, Haryanto L, Muhardin E. 2007. *Cara Cepat Mengembangkan Solusi Java Enterprise dengan Arsitektur MVC*. Jakarta: Bambumas.

Themistocleous, M. *et al*. *ERP Problems and Application Integration Issue: An Empirical Survey*. Departement of Information System and Computing, Brunel University.

Walls C, Breidenbach R. 2005. *Spring in Action.* United States: Manning.