APLIKASI *UML JAVA CODE GENERATOR* MENERAPKAN *DESIGN PATTERN*

[1]Siti Isnaini A. K., [2]Yonathan Ferry H., [3]Moch. Kautsar Sophan
[1,2,3]Program Studi Teknik Informatika
Universitas Trunojoyo Madura
Jl. Raya Telang, PO BOX 2, Kamal, Bangkalan – 69162
Email: [1]ainie1224@gmail.com, [2]yonathan.hendrawan@trunojoyo.ac.id,
[3]kautsar@if.trunojoyo.ac.id

Abstrak. *Sebuah aplikasi "UML Java Code Generator" dapat membantu developer dalam proses menerjemahkan UML ke dalam bentuk tulisan program. Untuk merancang aplikasi dengan baik dan terstruktur sehingga rancangan aplikasi berkualitas dan dapat mudah dimengerti, maka diterapkan design pattern. Design pattern dapat meningkatkan skalabilitas dan maintainabilitas dari aplikasi. Design pattern yang diterapkan adalah composite pattern sebagai desain pola struktural dan command pattern sebagai desain pola behavioral. Hasil penelitian adalah: (1) penerapan command pattern pada aplikasi dapat mendukung fungsionalitas undo dan redo yang dapat memudahkan user dalam menggunakan aplikasi (2) penerapan composite pattern bermanfaat karena kebutuhan terhadap dua atau lebih subclass dapat diwakili oleh satu superclass.*

Kata Kunci: UML generator, Design Pattern, Command Pattern, Composite Pattern

Dalam melakukan pendekatan dengan OOP (*Object Oriented Programming*) *developer* memerlukan UML (*Unified Modelling Language*) untuk menggambarkan model aplikasi. UML akan memudahkan developer ketika melanjutkan pada tahap penulisan program karena gambaran luas aplikasi sudah ada, sehingga *developer* dapat melanjutkan ke bagian detail.

Ketika *developer* menggunakan UML, *developer* memerlukan waktu yang lebih lama untuk melakukan pekerjaannya. Karena *developer* harus memodelkan gambaran aplikasi melalui UML lalu kemudian menerjemahkan UML ke dalam bentuk tulisan program (*coding*). Sebuah aplikasi "UML Java Code Generator" dapat membantu *developer* dalam proses penerjemahan UML ke dalam bentuk *coding*.

Tahap Perancangan pada aplikasi merupakan tahap penting yang menentukan kualitas suatu aplikasi. Untuk merancang aplikasi dengan baik dan terstruktur sehingga rancangan aplikasi berkualitas dan dapat mudah dimengerti, maka penulis akan menerapkan *design pattern*. *Design pattern* adalah solusi umum dalam menangani masalah perancangan aplikasi. *Design pattern* dapat meningkatkan skalabilitas dan maintainabilitas dari aplikasi. Dalam merancang aplikasi ini, akan diterapkan *composite pattern* sebagai desain pola struktural dan *command pattern* sebagai desain pola behavioral.

Penerapan Pola Desain Untuk Perancangan Aplikasi Stasiun Cuaca Nirkabel

Lintang Dwi Febriani [1] melakukan penelitian dengan menerapkan *design pattern* pada aplikasi stasiun cuaca nirkabel agar desain aplikasi tersebut berkualitas dan maintainabilitas. *Design pattern* yang digunakan adalah *Abstract Factory* dan *Mediator*.

Abstract Factory digunakan sebagai solusi untuk menyelesaikan permasalan bagaimana membuat *interface* untuk sekumpulan objek yang saling berelasi tanpa secara eksplisit menspesifikasi kelas. Desain *Abstract Factory* ini digunakan pada desain struktural SensorManager pada aplikasi.

Mediator digunakan sebagai solusi dari permasalahan bagaimana membuat objek yang mengenkapsulasi sekumpulan objek yang saling berinteraksi. Desain *Mediator* ini digunakan pada desain *behavioral* aplikasi dengan tujuan mengurangi tingkat ketergantungan antara kelas utama (MainWindow) dan kelas model (Parsing dan SensorMgr).

Effective Apply Of Design Pattern In Database-Based Application Development

Hao Dai [2] dari China telah melakukan penelitian tentang penerapan *design pattern* pada pengembangan aplikasi *database-based* secara efektif. Dalam pengembangan aplikasi ini juga digunakan OOP (*Object Oriented Programming*) untuk mengoptimalkan *code* terkait dengan struktur dan maintainabilitas seluruh aplikasi. Karena tanpa optimasi yang tepat, maka akan meningkatkan biaya perawatan aplikasi.

Design pattern yang digunakan diantaranya *Adapter Pattern* untuk memisahkan akses data dan bagian *code logic*, *Factory Pattern* untuk membangun objek, *Decorator Pattern* untuk mengembangkan fungsi *driver database*, dan *Command pattern* untuk dapat mencapai *retrying connection database* secara otomatis.

Dari penelitian ini terbukti bahwa *design pattern* membuat struktur aplikasi mudah dimengerti, sehingga dapat meningkatkan skalabilitas dan maintainabilitas aplikasi.

A Goal-Oriented Approach For Representing And Using Design Pattern

Luca Sabatucci et al [3] telah mengusulkan dokumentasi *pattern goal-oriented* yang menekankan pada informasi keputusan yang relevan dalam penelitiannya. Penelitian ini menyajikan notasi visual yang memvisualisasikan konteks, solusi *alternative* dan konsekuensi. Penelitian ini juga memperkenalkan sebuah proses *reuse* sistematis dimana penggunaan *pattern goal-oriented* yang dapat membantu praktisi dalam memilih dan menyesuaikan *design pattern*.

Design pattern yang digunakan diantaranya *Mediator Pattern*, *Embassy Pattern*, *Block Data Pattern*, *State Pattern*, *Strategy Pattern*, *Flyweight Pattern*, dan *Broker Pattern*. Eksperimen dari penelitian ini mengungkapkan bahwa solusi yang baik adalah dengan menerapkan *pattern* ketika menyusun requirement.

Design Pattern

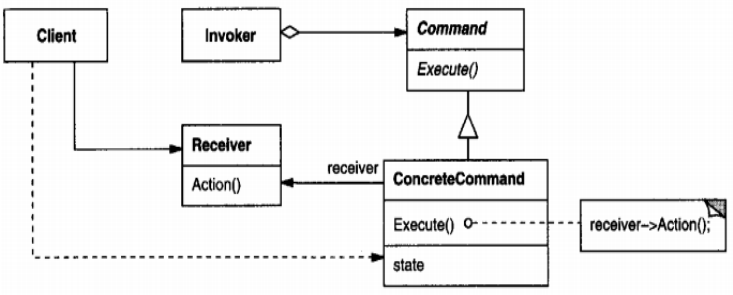
Design pattern adalah sebuah solusi umum untuk menyelesaikan masalah-masalah umum yang ditemukan dalam desain perangkat lunak. Istilah *design pattern* awalnya dikemukakan oleh Christopher Alexander dalam bukunya yang berjudul *A Pattern Language* yaitu "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, insuch a way that you can use this solution a million times over, without everdoing it the same way twice*." [4]

Menurut The Gang of Four [5] *design pattern* dibedakan menjadi 3 bagian berdasarkan fungsi dan kegunaannya, yaitu:

1. *Creational Patterns*, yaitu *pattern* yang fokus pada penciptaan objek. *Pattern* ini berkisar seputar objek mana yang diciptakan, siapa yang menciptakannya, serta berapa banyak objek yang diciptakan. *Creational pattern* ini diantaranya adalah *Abstract Factory*, *Factory Method*, dan *Singleton*.
2. *Structural Patterns*, yaitu pattern yang memberikan komposisi sebuah kelas/objek. *Structural pattern* ini diantaranya yaitu *Bridge pattern, Composite pattern, Decorator pattern, Façade*, dan *Flyweight pattern*.
3. *Behavioral Patterns*, yaitu *pattern* yang mengatur tingkah laku (*behavior*), interaksi atau fungsi dari suatu kelas/objek. *Behavioral pattern* ini diantaranya adalah *Command pattern, Iterator pattern, Observer pattern, Memento pattern, Strategy pattern*, dan *Template method.*

Command Pattern

Command pattern membungkus sebuah permintaan ke dalam sebuah objek. *Command pattern* mendukung operasi *undo-redo*. Adapun struktur kelas dari *command pattern* dapat dilihat pada gambar di bawah ini.



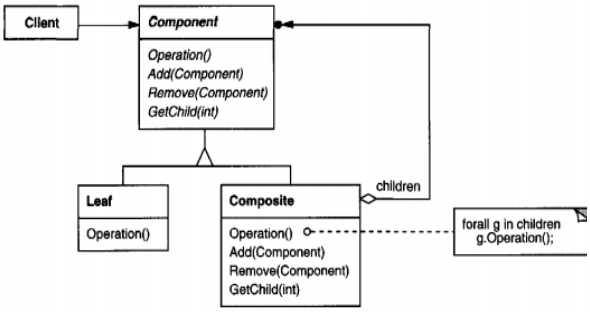
Gambar 1. Struktur Kelas *Command Pattern* [5]

Command pattern dapat digunakan ketika:

1. Menjalankan permintaan pada waktu yang berbeda-beda.
2. Aplikasi hendak menerapkan operasi *undo-redo.*

Composite Pattern

Composite pattern menyusun objek-objek ke dalam struktur pohon untuk mewakili hirarki seluruh atau sebagian. Dengan *composite* aplikasi dapat memperlakukan objek tunggal maupun sekumpulan objek dengan cara yang sama. Adapun struktur kelas dari *composite pattern* dapat dilihat pada gambar dibawah ini.



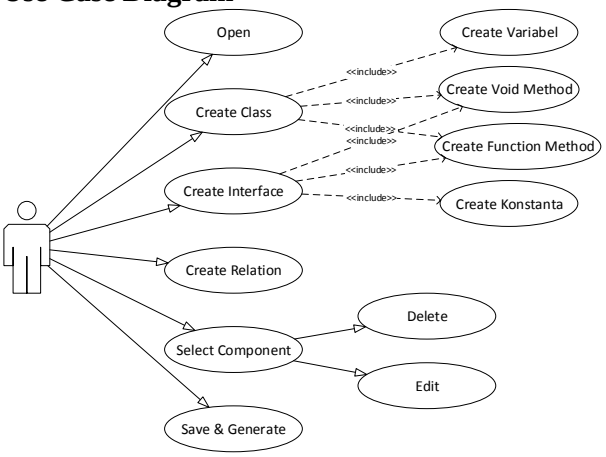
Gambar 2. Struktur Kelas *Composite Pattern* [5]

Composite pattern dapat digunakan ketika:

1. Aplikasi hendak merepresentasikan hirarki secara menyeluruh ataupun sebagian.
2. Aplikasi hendak memperlakukan objek tunggal maupun komposisi dengan mengesampingkan perbedaan.

I. Metodologi

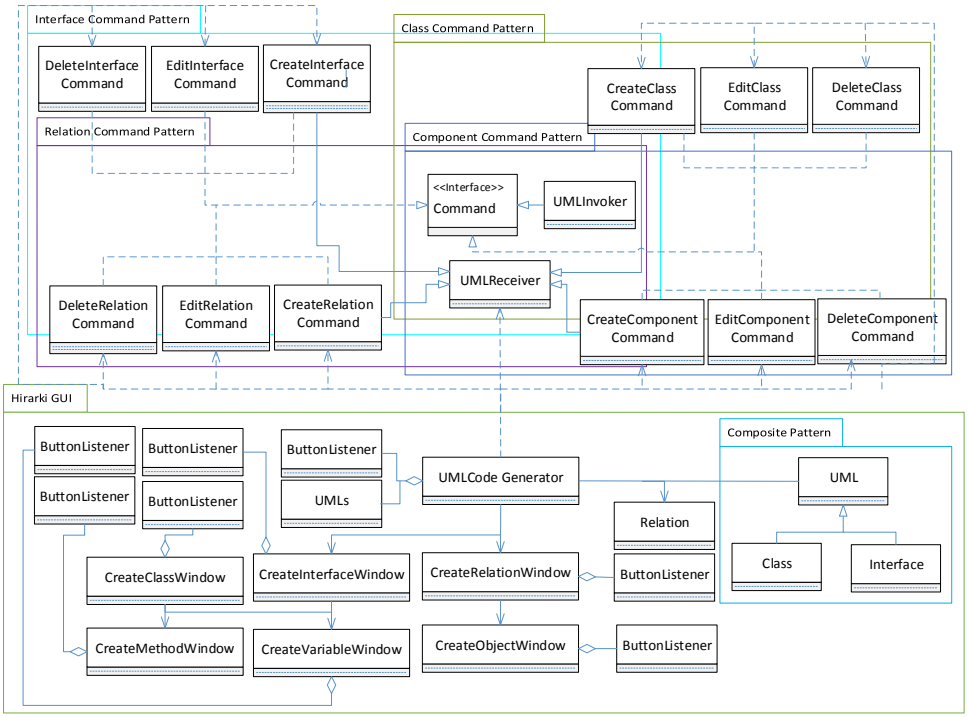
Use Case Diagram



Gambar 3. Use Case Diagram Aplikasi

Dari *use case diagram* pada gambar diatas dapat dilihat bahwa pada aplikasi "UML Java Code Generator" user dapat mengakses menu *Open*, *Create Class*, *Create Interface*, *Create Relation*, Select, dan *Save* and *Generate*. Pada menu *Create Class* terdapat *submenu Create Variabel*, *Create Void*, dan *Create Function*. Pada menu *Create Interface*, terdapat *submenu Create Void* dan *Create Function*. Dan pada menu *Select*, user dapat menghapus dan mengubah UML yang telah di-*select*.

Rancangan Total Kelas

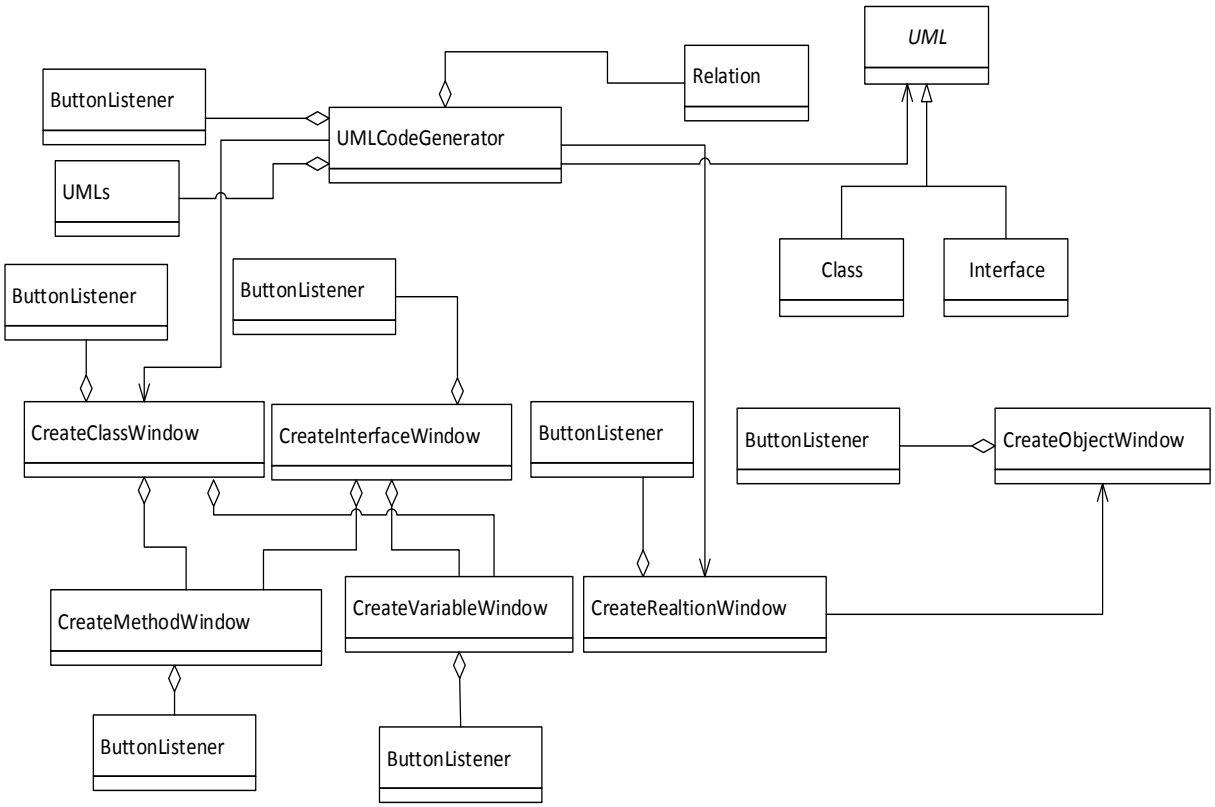


Gambar 4. Rancangan Total Kelas

Pada gambar diatas dapat dilihat bahwa dari rancangan total kelas terdapat beberapa rancangan yang saling beririsan. Beberapa rancangan tersebut adalah rancangan hirarki GUI, rancangan *composite pattern*, dan rancangan *command pattern*. Untuk masing-masing rancangan penjelasan diuraikan pada uraian berikutnya.

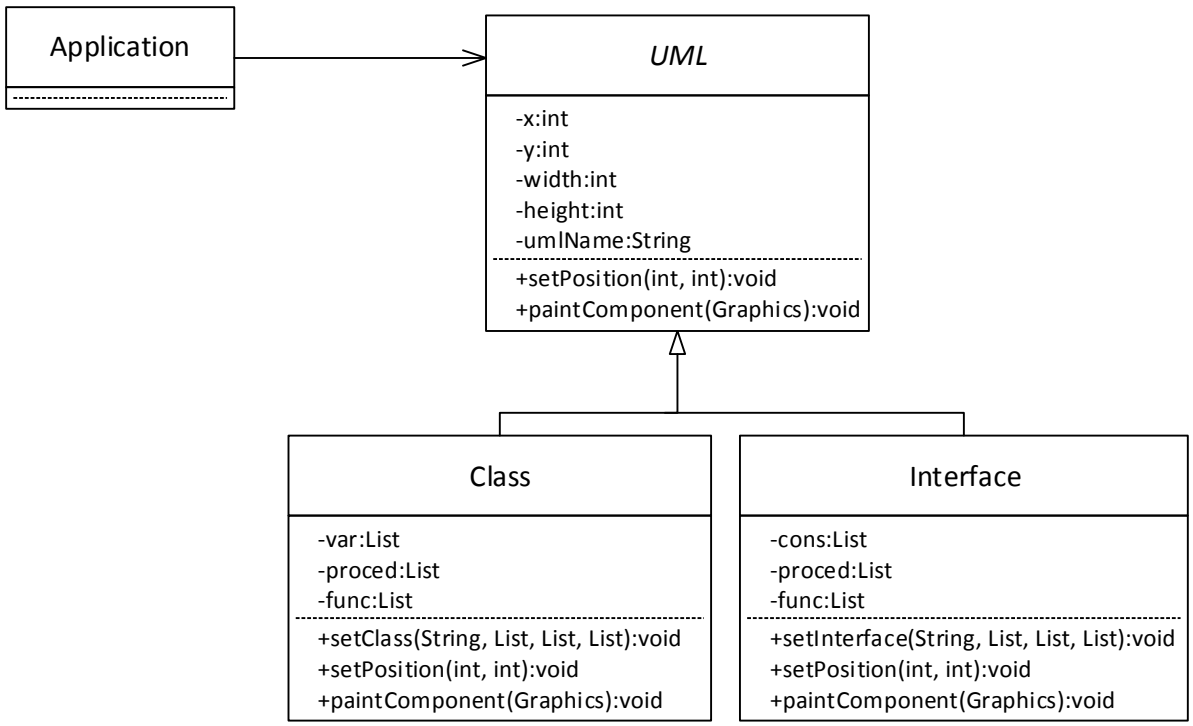
Rancangan Hirarki GUI

Rancangan hirarki GUI adalah rancangan dari kelas-kelas yang bertugas sebagai tampilan GUI. Rancagan hirarki GUI terdiri kelas `UMLCodeGenerator`, `CreateClassWindow`, `CreateInterfaceWindow`, `CreateRelationWindow`, `CreateVariableWindow`, `CreateMethodWindow`, `CreateObjectWindow`, dan rancangan *composite pattern*. Kelas `UMLCodeGenerator` merupakan window utama aplikasi. Kelas `CreateClassWindow` merupakan kelas yang menangani window "Create Class". Kelas `CreateInterfaceWindow` merupakan kelas yang menangani window "Create Interface". Kelas `CreateRelationWindow` merupakan kelas yang menangani window "Create Relation". Kelas `CreateVariableWindow` merupakan kelas yang menangani window "Create Variable". Kelas `CreateMethodWindow` merupakan kelas yang menangani window "Create Method". Kelas `CreateObjectWindow` merupakan kelas yang menangani window "Create Object". Dan rancangan composite pattern akan dijelaskan pada rancangan berikutnya.



Gambar 5. Rancangan Hirarki GUI

Rancangan *Composite Pattern*

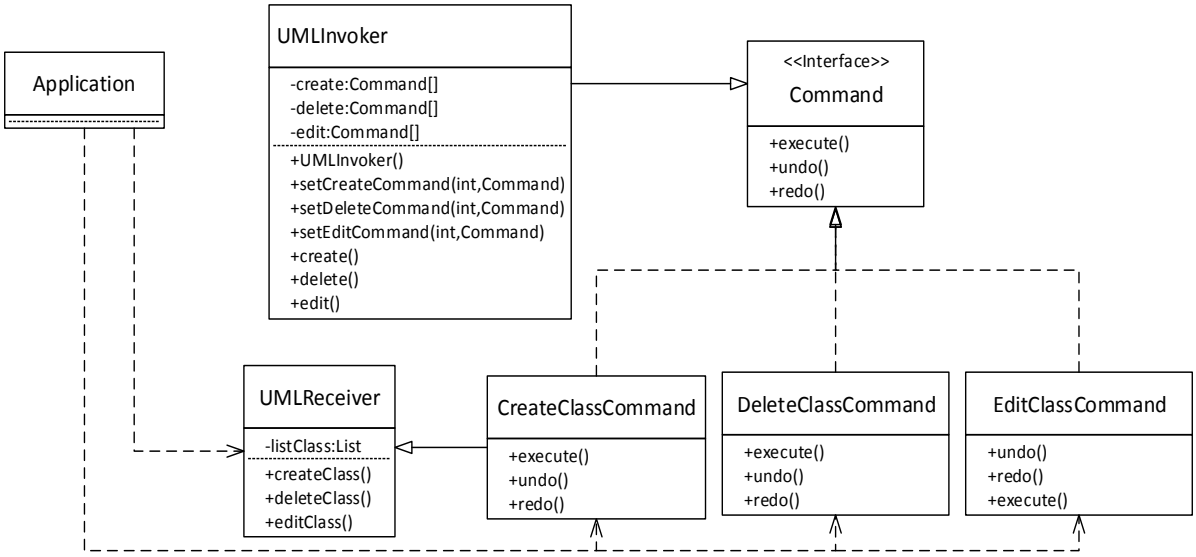


Gambar 6. Rancangan *Composite Pattern*

Rancangan *composite pattern* terdiri dari kelas `UML`, `Class`, dan `Interface`. Kelas `UML` merupakan *superclass* sedangkan kelas `Class` dan `Interface` merupakan *subclass* yang bertugas menangani penggambaran *UML diagram* pada aplikasi. Dengan rancangan *composite pattern*, kebutuhan terhadap *subclass* dapat diwakili oleh *superclass*.

Rancangan *Command Pattern*

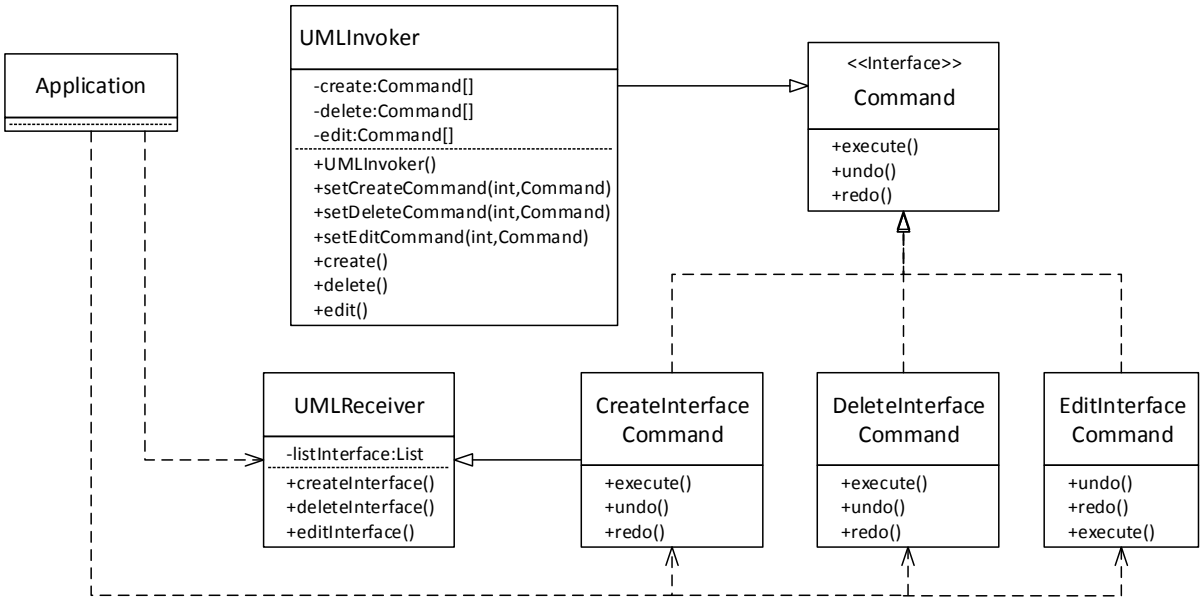
Rancangan *command pattern* diterapkan untuk membungkus perintah *create*, *delete*, *edit*. Rancangan *command pattern* untuk perintah *create*, *delete*, dan *edit* pada *Class* dapat dilihat pada gambar berikut.



Gambar 7. Rancangan *Command Pattern* untuk perintah-perintah terhadap *Class*

Dari gambar diatas dapat dilihat bahwa rancangan terdiri atas kelas `UMLInvoker`, `UMLReceiver`, `CreateClassCommand`, `DeleteClassCommand`, `EditClassCommand`, dan *interface* `Command`. Kelas `UMLInvoker` merupakan *invoker* dari *command pattern* yaitu sebagai *remote* yang terdiri dari aksi-aksi *create*, *delete*, dan *edit*. Kelas `UMLReceiver` merupakan *receiver* dari *command pattern*. Kelas `CreateClassCommand`, `DeleteClassCommand`, dan `EditClassCommand` merupakan kelas-kelas yang menerapkan *interface* `Command`. Kelas-kelas inilah yang akan membungkus perintah *create*, *delete*, dan *edit* terhadap *Class*.

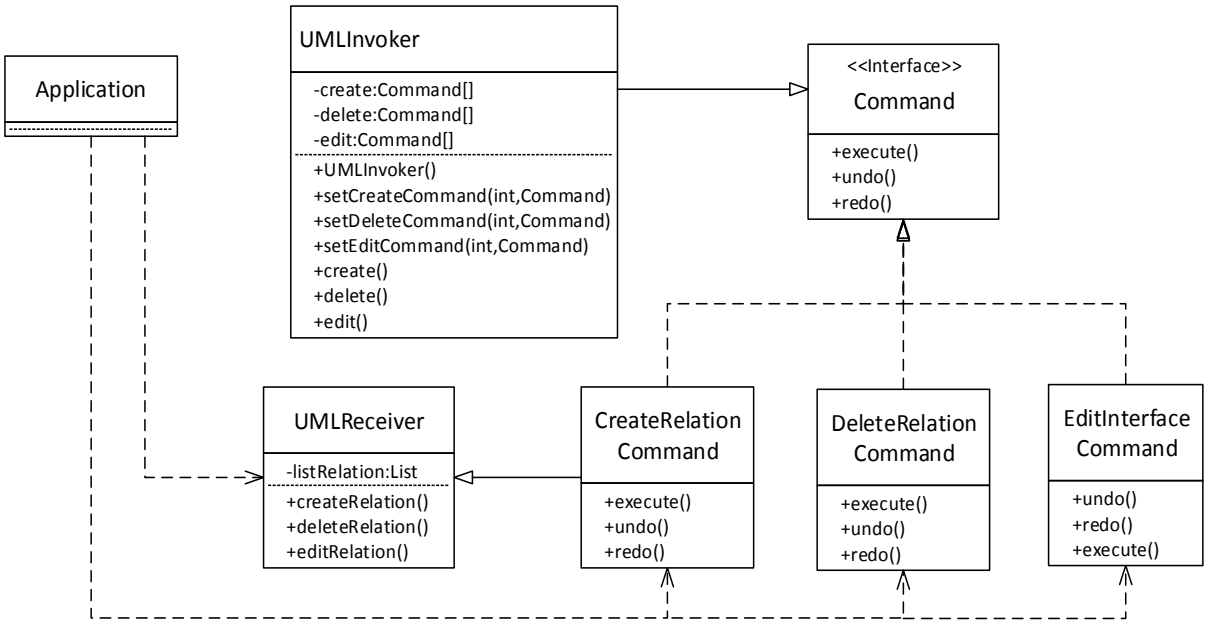
Rancangan *command pattern* untuk perintah *create*, *delete*, dan *edit* pada *Interface* dapat dilihat pada gambar berikut.



Gambar 8. Rancangan *Command Pattern* untuk perintah-perintah terhadap *Interface*

Dari gambar diatas dapat dilihat bahwa rancangan terdiri atas kelas `UMLInvoker`, `UMLReceiver`, `CreateInterfaceCommand`, `DeleteInterfaceCommand`, `EditInterfaceCommand`, dan *interface* `Command`. Kelas `UMLInvoker` merupakan *invoker* dari *command pattern* yaitu sebagai *remote* yang terdiri dari aksi-aksi *create*, *delete*, dan *edit*. Kelas `UMLReceiver` merupakan *receiver* dari *command pattern*. Kelas `CreateInterfaceCommand`, `DeleteInterfaceCommand`, dan `EditInterfaceCommand` merupakan kelas-kelas yang menerapkan *interface* `Command`. Kelas-kelas inilah yang akan membungkus perintah *create*, *delete*, dan *edit* terhadap *Interface*.

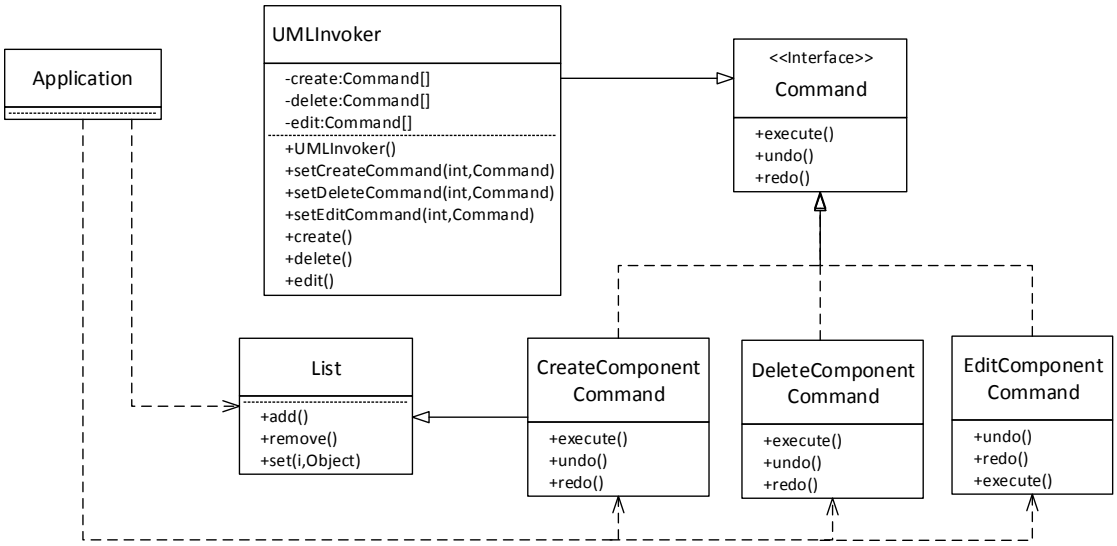
Rancangan *command pattern* untuk perintah *create*, *delete*, dan edit pada *Relation* dapat dilihat pada gambar berikut:



Gambar 9. Rancangan *Command Pattern* untuk perintah-perintah terhadap *Relation*

Dari gambar diatas dapat dilihat bahwa rancangan terdiri atas kelas `UMLInvoker`, `UMLReceiver`, `CreateRelationCommand`, `DeleteRelationCommand`, `EditRelationCommand`, dan *interface* `Command`. Kelas `UMLInvoker` merupakan *invoker* dari *command pattern* yaitu sebagai *remote* yang terdiri dari aksi-aksi *create*, *delete*, dan *edit*. Kelas `UMLReceiver` merupakan *receiver* dari *command pattern*. Kelas `CreateRelationCommand`, `DeleteRelationCommand`, dan `EditRelationCommand` merupakan kelas-kelas yang menerapkan *interface* `Command`. Kelas-kelas inilah yang akan membungkus perintah *create*, *delete*, dan *edit* terhadap *Relation*.

Rancangan *command pattern* untuk perintah *create*, *delete*, dan *edit* pada *Component* dapat dilihat pada gambar berikut. Dari gambar rancangan dapat dilihat bahwa rancangan terdiri atas kelas `UMLInvoker`, `UMLReceiver`, `CreatComponentCommand`, `DeleteComponentCommand`, `EditComponentCommand`, dan *interface* `Command`. Kelas `UMLInvoker` merupakan *invoker* dari *command pattern* yaitu sebagai *remote* yang terdiri dari aksi-aksi *create*, *delete*, dan *edit*. Kelas `UMLReceiver` merupakan *receiver* dari *command pattern*. Kelas `CreateComponentCommand`, `DeleteComponentCommand`, dan `EditComponentCommand` merupakan kelas-kelas yang menerapkan *interface* `Command`. Kelas-kelas inilah yang akan membungkus perintah *create*, *delete*, dan *edit* terhadap *Component*.



Gambar 10. Rancangan *Command Pattern* untuk perintah-perintah terhadap *Component*

II. Hasil dan Pembahasan

Implementasi Dan Uji Coba *Composite Pattern*

Dalam proses pembuatan relasi, dibutuhkan 2 bagan UML class atau UML relasi yang akan berelasi. Kelas yang bertugas membuat UML Class adalah kelas Class, sedangkan kelas yang bertugas membuat UML Interface adalah kelas Interface. Dengan menggunakan *composite pattern*, kebutuhan terhadap kelas Class dan kelas Interface tersebut dapat diwakili oleh kelas UML yang merupakan superclass dari kelas Class dan kelas Interface. Sehingga tidak perlu membangun objek dari kelas Class dan kelas Interface untuk menyimpan bagan UML tersebut. Dengan membuat objek dari kelas UML, objek tersebut dapat dipakai untuk menyimpan bagan UML Class dari kelas Class atau bagan UML Interface dari kelas Interface.

Pada kelas CreateRelationWindow dibuat 2 objek dari kelas UML untuk menyimpan bagan pertama dan kedua.

```
public class CreateRelationWindow {
    UML node1, node2;
    public CreateRelationWindow (UMLCodeGenerator objUCG) {
    }
}
```

Dan fungsi untuk membangun relasi menggunakan kedua objek dari kelas UML tersebut. Fungsi tersebut adalah sebagai berikut.

```
    void createRelation(UML node1, UML node2, int relation, LinkedList<String>
objects){
        Relation newRel = new Relation(node1, node2, relation, objects);
        CreateRelationCommand create = new CreateRelationCommand(receiver,
newRel);
        invoker.setCreateCommand(2, create);
        invoker.create(2);
        btnUndo.setEnabled(true);
        btnRedo.setEnabled(false);
    }
```

Dari fungsi tersebut dapat dilihat bahwa untuk membangun relasi yaitu dengan membangun objek dari kelas Relation, dimana kelas tersebut yang bertugas menggambar relasi. Dan kelas Relation juga membutuhkan kedua objek dari kelas UML tersebut. Detai kelas Relation dapat dilihat pada *coding* berikut.

```
      public class Relation extends JPanel implements Serializable{
          UML node1, node2;
          int relationType;
          LinkedList<String> Objects;

          public Relation(UML node1, UML node2, int relationType,
LinkedList<String> objects) {
              this.node1 = node1;
              this.node2 = node2;
              this.relationType = relationType;
              this.Objects=objects;
          }
      }
```

Implementasi Dan Uji Coba *Command Pattern*

Tabel 1. Uji Coba Command Pattern

SKENARIO	PROSES UJI COBA	HASIL
Create Variable	Uji coba dilakukan dengan membuat sebuah variabel kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Delete Variable	Uji coba dilakukan dengan menghapus sebuah variabel kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Edit Variable	Uji coba dilakukan dengan meng-*edit* sebuah variabel kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Create Method	Uji coba dilakukan dengan membuat sebuah *method* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Delete Method	Uji coba dilakukan dengan menghapus sebuah *method* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Edit Method	Uji coba dilakukan dengan meng-*edit* sebuah *method* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Create Class	Uji coba dilakukan dengan membuat sebuah kelas kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Delete Class	Uji coba dilakukan dengan menghapus sebuah kelas kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Edit Class	Uji coba dilakukan dengan meng-*edit* sebuah kelas kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Create Interface	Uji coba dilakukan dengan membuat sebuah *interface* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Delete Interface	Uji coba dilakukan dengan menghapus sebuah *interface* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Edit Interface	Uji coba dilakukan dengan meng-*edit* sebuah *interface* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Create Relation	Uji coba dilakukan dengan membuat sebuah *relation* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Delete Relation	Uji coba dilakukan dengan menghapus sebuah *relation* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL
Edit Relation	Uji coba dilakukan dengan meng-*edit* sebuah *relation* kemudian dilakukan proses *undo* dan *redo*.	BERHASIL

Pada tabel diatas dapat dilihat bahwa beberapa perintah yang menerapkan *command pattern* berhasil menggunakan fungsional *undo* dan *redo* yang didukung oleh *command pattern*.

III. Simpulan

Setelah menyelesaikan perancangan dan pembuatan aplikasi "UML Java Code Generator" dengan melalui serangkaian pengujian, maka dapat disimpulkan bahwa :

1. Dalam membangun aplikasi "UML Java Code Generator" melalui beberapa tahap yaitu : pengumpulan informasi dan studi

literatur, perencanaan, pembuatan, dan uji coba.

2. Penerapan *command pattern* pada aplikasi ini dapat mendukung fungsionalitas *undo* dan *redo* yang dapat memudahkan user dalam menggunakan aplikasi.
3. Penerapan *composite pattern* bermanfaat bagi penulis karena dengan penerapan *composite pattern* kebutuhan terhadap dua atau lebih *subclass* dapat diwakili oleh *superclass*.

IV. Daftar Pustaka

[1] Febriani, LD. Penerapan Pola Desain Untuk Perancangan Aplikasi Stasiun Cuaca Nirkabel. *Jurnal Pusat Penelitian Informatika*, Lembaga Ilmu Pengetahuan Indonesia, Bandung. 2012.

[2] Dai, H. Effective Apply of Design Pattern in Database-based Application Development *Jurnal Department of Computer Science*, Guangdong Posts & Telecom Vocational Technology College, Guangzhou China. 2012.

[3] Sabatucci, Luca., Cossentino, Massimo., Susi, Angelo. A goal-oriented approach for representing and using design patterns *Jurnal The Journal of Systems and Software*, Italy. 2015.

[4] Alexander, Christopher., Ishikawa, Sara., Silverstain, Murray. (1977). "*A Pattern Language : Towns, Buildings, Construction*", New York : Oxford University Press.

[5] Gamma, Erich., Helm, Richard., Johnson, Ralph., Vlissides, John. *Design Patterns : Elements of Reusable Object-Oriented Software*. New York : Addison-Wesley Publishing Company. (1995)

[6] Freeman, Eric., Freeman, Elisabeth., Sierra, Kathy., Bates, Bert. *Head First Design Patterns.* United State of America : O'Reilly Media, Inc. 2004.

[7] Kusnawi., Penerapan Design Patterns Untuk Perancangan Berbasiskan Objek Oriented. *Jurnal* 2010.

[8] Tegarden, David., Dennis, Alan., Wixom, Barbara Haley. *System Analysis and Design with UML*. Singapore : John Wiley & Sons, Inc. 2013.