Pengenalan Design Pattern C# Dan Implementasi Proxy Pattern Sebagai Keamanan Software

Tugas Akhir Mata Kuliah Keamanan Perangkat Lunak (EL5215)
Sigit Ari Wijanarko
(23214352)
Magister Teknik Elektro
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2016

ABSTRAK

***Programmer* yang sukses memiliki dua hal utama yaitu bahasa pemrograman yang baik dan *design patterns*. *Design pattern* pada *software* pertama kali telah diperkenalkan oleh Erich Gamma, Richard Helm, Ralph Johnson,and John Vlissides's. *Patterns* ini mampu mengatasi permasalahan pada pemprograman berorientasi obyek (OOP) pada c++. Sejak saat itu mulai berkembang *pattern* di java, visual basic dan C#. Tujuan dari *design pattern* adalah membuat kode program menjadi mudah, bersih dan aman.**

Pada makalah ini akan membahas secara singkat tentang *design pattern* dan membuktikan kelibihan *pattern*, dengan memilih salah satu implementasi dari *design pattern* yaitu *proxy pattern*. *Proxy pattern* memungkinkan untuk mengontrol suatu obyek untuk membuat dan mengakses obyek lain. *Proxy pattern* memiliki fungsi sebagai gerbang dalam mengakses objek lain. Dapat diilustrasikan, misalkan ketika membeli makanan di restoran mewah. Dalam membeli makanan di restoran yang dilakukan adalah melihat menu, memesan, dan membayar makanan. Disisi lain, karena masakan bersifat rahasia maka pembeli tidak dapat melihat cara memasak. Bahkan semua restoran atau tempat makan, pembeli tidak dapat ikut andil dalam memasak karena dapat mempengaruhi kwalitas rasa masakan.

Kata kunci : ***design pattern, proxy pattern, interface, access modifiers, pattern C#.***

1. PENDAHULUAN

Perkembangan *software* sangat cepat diiringi dengan meningkatnya kekompleksitasnya *software* tersebut., dari pemprograman struktural ke pemrograman berorentasi obyek. Pemrograman terstruktur adalah pengorganisasian dari *method* dan *coding* program yang dapat mudah dipahami dan modifikasi, sedangkan pemrograman berorentasi obyek (OOP) terdiri dari satu set obyek, yang dapat bervariasi secara dinamis, dan yang saling berhubungan satu sama lain [1]. Pemprograman berorientasi obyek membuat program lebih intuitif untuk merancang, lebih cepat untuk mengembangkan, mudah untuk modifikasi, dan lebih mudah untuk memahami [1].

Software yang kompleks membutuhkan desain karena tidak hanya satu orang yang menggunakan, tetapi memungkinkan untuk digunakan banyak pengguna dengan berbagai kepentingan [3]. *Software* dirancang dengan dibagi-bagi menjadi berbagai bagian, bahkan dalam mengembangan *software* dilakukan oleh tim yang terdiri dari beberapa orang pengembang dan program yang dikembangkan sebelumnya. Bagian-bagian tersebut akan saling ketergantungan satu sama lain.

Banyak yang mengatakan, menjadi programmer merupaka suatu bakat sejak lahir, karena mampu duduk berjam-jam untuk menyelesaikan masalah dan membuat semu menjadi lebih baik [4]. Namun teknik dalam membuat code program yang unggul dan praktek dalam membuat kode yang unggul merupakan acuhan dari programer professional. Teknik dalam membuat kode merupakan teknik dalam menggunaan *design pattern*. Tujuan dari design pattern adalah membuat kode program menjadi mudah, bersih dan aman [4]. Telah dijelaskan, dalam pembuatan program dapat dibagi menjadi beberapa bagian. Bagian-bagian ini memiliki funsi masing-masing yang dipanggil sesuai dengan kebutuhan. Namun dalam penggunaan fungsi tidak semua fungsi dapat diakses, ini dilakukan untuk keamanan dari bagian software tersebut. Misalnya pada aplikasi sosial media, antara *admin* dan *user* yang mempunyai hak akses *database* yang berbeda. *User* memiliki hak akses hanya melihat *user* lain, sedangkan *admin* selain bisa melihat user juga mampu menghapus *user*. Didalam makalah ini

akan dijelaskan secara singkat *design pattern* pada bahasa pemprograman C#, khususnya untuk C# 3.0 dan akan membuktikan keuntungan *design pattern* yaitu sebagai *security*.

2. TENTANG DESIGN PATTERN

Pertama kali diperkenalkan oleh seorang arsitek yaitu Christopher Alexander. Menurut Christopher Alexander *design pattern* adalah pengggunaan kembali dari solusi untuk mendesain masalah [7]. Christopher Alexander mengatakan, setiap pola menggambarkan masalah yang terjadi berulang-ulang dan kemudian menjelaskan inti dari solusi untuk masalah itu, sehingga dapat digunakan sebagai solusi. Meskipun Alexander berbicara tentang pola di gedung-gedung dan kota-kota, apa yang dia katakan benar tentang desain *pattern* berorientasi obyek. Dengan adanya *design pattern* maka masalah yang sama dapat diselesaikan dengan mengacu *pattern* yang sama [2]. *Design pattern* ini berkembang keberbagi disiplin ilmu, terutama dalam bidang ilmu komputer [7].

Dalam bidang ilmu komputer *design pattern* telah diperkenalkan oleh Erich Gamma, Richard Helm, Ralph Johnson,and John Vlissides's [4]. *Design pattern* membuat lebih mudah untuk menggunakan kembali desain yang sukses dan arsitekturnya. Mengungkapkan teknik yang terbukti membuat lebih mudah untuk pengembang sistem baru. *Design pattern* membantu untuk memilih alternatif desain yang membuat sistem dapat digunakan kembali dan menghindari alternatif yang tidak tepat. *Design pattern* bahkan dapat meningkatkan dokumentasi dan pemeliharaan sistem [2].

Secara umum, *pattern* memiliki empat elemen penting:

- *Pattern Name*
  Digunakan sebagai acuhan dalam menggambarkan masalah desain, solusi, dan konsekuensi yang biasanya menggunakan satu atau dua kata. Nama *pattern* mempunyai kedudukan level lebih tinggi dari pada abstrak. Hal itu membuat mudah dalam berpikir tentang desain [2]. Programmer akan mudah dalam menggunakan *pattern* mana yang sesuai dengan permasalahanya. Misalkan *pattern decoration*, maka akan berpikir bahwa fungsi *pattern* ini digunakan untuk dekorasi obyek. Dengan menggunakan *pattern decoration* maka obyek *image* dapat dimodifikasi, misalnya dengan menambahkan *frame*.
- *Problem* yang menjelaskan ketika menggunakan *pattern*.
  Ini menjelaskan suatu *problem* dan *context*. Memungkin untuk menjelaskan struktur *class* atau *objec*t yang memiliki desain yang fleksible [2]. Misalkan memerlukan fitur *Generics* C#, yang memungkinkan untuk membuat *type* data menjadi fleksibel.
- *Solution* menjelaskan unsur-unsur yang membentuk *pattern*, hubunga antar unsur, *responsibilities*, and kolaborasi. *Pattern* seperti *template* yang mampu digunakan dibanyak situasi yang berbeda [2].
- *Consequences* merupakan hasil penerapan *pattern*.
  Hal ini sering terlupakan dalam penerapan penggunaan *design pattern*, namun ini sangat penting digunakan untuk mengevaluasi *pattern* yang digunakan yaitu memahami kerugian dan keuntungan *pattern* yang digunakan bahkan dapat digunakan untuk alternatif dari sebuah *pattern* [2].

Ada banyak *design patterns* yang sudah diakui kemampuannya, diterima dan diaplikasikan oleh banyak praktisi. *Design Patterns* yang cukup populer adalah yang diperkenalkan The Gang of Four (GoF) - Erich Gamma, Richard Helm, Ralph Johnson dan John Vlissides. Dalam The Gang of Four (GoF) terdapat 23 *Pattern* yang dibagi menjadi 3 kelompok besar yaitu *creational*, *structural*, and *behavioral* [2].

- **Creational Patterns** (cara class/object di-inisiasi).
  1. *Abstract Factory* (*Creates an instance of several families of classes*)
  2. *Builder* (*Separates object construction from its representation*)
  3. *Factory Method* (*Creates an instance of several derived classes*)
  4. *Prototype* (*A fully initialized instance to be copied or cloned*)
  5. *Singleton* (*A class of which only a single instance can exist*)
- ***Structural Patterns*** (struktur/relasi antar object/*class*)
  1. *Adapter (Match interfaces of different classes)*
  2. *Bridge (Separates an object's interface from its implementation)*
  3. *Composite (A tree structure of simple and composite objects)*
  4. *Decorator (Add responsibilities to objects dynamically)*
  5. *Facade (A single class that represents an entire subsystem)*
  6. *Flyweight (A fine-grained instance used for efficient sharing)*
  7. *Proxy (An object representing another object)*
- ***Behavioral Patterns*** (tingkah laku atau fungsi dari class/object.)
  1. *Chain of Responbility (A way of passing a request between a chain of objects)*
  2. *Command (Encapsulate a command request as an object)*

3. *Interpreter (A way to include language elements in a program)*
4. *Iterator (Sequentially access the elements of a collection)*
5. *Mediator (Defines simplified communication between classes)*
6. *Mementto (Capture and restore an object's internal state)*
7. *Observer (A way of notifying change to a number of classes)*
8. *State (Alter an object's behavior when its state changes)*
9. *Strategy (Encapsulates an algorithm inside a class)*
10. *Template Method (Defer the exact steps of an algorithm to a subclass)*
11. *Visitor (Defines a new operation to a class without chang*

.

3. DESAIN *PATTERN* MENYELESAIAN MASALAH DESAIN

Desain *pattern* banyak memecahkan masalah untuk persoalan desainer pemprograman berorentasi obyek, dan dilakukan dengan banyak cara yang berbeda [2]. Berikut adalah beberapa dari masalah dan bagaimana *design pattern* menyelesaikannya.

- **Menemukan *Appropriate Objects***

  Pemprograman berorientasi obyek yang terdiri dari obyek-obyek terdiri dari *packages* data dan *procedures* dari operasi data tersebut. *Procedures* biasanya disebut *method* atau operasi. Sebuah obyek melakukan operasi ketika menerima permintaan (atau pesan) dari klien.

  Requests murupakan satu-satunya cara untuk mendapatkan obyek dalam menjalankan operasi. Operasi adalah satu-satunya cara untuk mendapatkan data internal obyek. Karena pembatasan ini, *state* internal obyek dikatakan *encapsulated*, yaitu tidak dapat diakses secara langsung dan representasi tidak terlihat dari luar object.

  Bagian yang sulit pemprograman berorientasi obyek adalah *decomposing* (menguraikan) sistem menjadi object. Ini sulit karena banyak faktor yang terlibat. *encapsulation, granularity, dependency, flexibility, performance, evolution, reusability,* dan yang lainnya.

  Design patterns membantu untuk mengidentifikasi abstraksi kurang jelas dan obyek-obyek yang dapat digambarkan. Misalnya, obyek yang mewakili suatu proses atau algoritma tidak terjadi secara wajar, namun object tersebut adalah bagian penting dari desain yang fleksibel. Strategi *pattern* menjelaskan bagaimana menerapkan pertukaran algoritma yang tepat. *State pattern* mewakili masing-masing *state* dari suatu *entity* sebagai object.

- **Menentukan *Object Granularity***

  Obyek dapat sangat berfariasi dalam ukuran dan jumlah. Dapat mewakili semua lapisan bawah ke hardware atau semua lapisan atas sampai ke seluruh aplikasi.

  Facade pattern menjelaskan cara untuk mewakili subsistem lengkap sebagai obyek dan *Flyweight pattern* menjelaskan cara mendukung sejumlah besar object di *granularities* terbaik. *Design patterns* lainnya menjelaskan cara-cara khusus menguraikan sebuah obyek ke obyek yang lebih kecil. *Abstract Factory* dan *Builder* berfungsi untuk membuat object lain.

- **Menentukan *Interfaces* Object**

  Semua operasi dinyatakan oleh nama operasi, object membutuhkan parameter dan operasi mengembalikan nilai (*value*). Hal ini dikenal sebagai *signature* operasi. *Signature* didefinisikan sebagai operasi object yang disebut sebagai *interface* ke object. obyek *Interface* mencirikan set lengkap request (permintaan) yang dikirim ke obyek. Setiap *signature* yang cocok maka akan dapat dikirim ke obyek.

  Type merupakan nama yang digunakan untuk menunjukan *interface* tertentu. Sebuah obyek mungkin memiliki banyak jenis, dan banyak obyek yang berbeda dapat berbagi type. *Interface* dapat berisi *interface* lain sebagai subset.

  Interface merupakan dasar dalam sistem berorientasi obyek. Obyek yang dikenal hanya melalui *interface*, tidak ada cara untuk mengetahui tentang suatu obyek atau untuk meminta melakukan sesuatu tanpa melalui *interface*. *Interface* obyek dapat menggunakan implementasi yang berbeda, sesuai dengan kebutuhan. Itu berarti dua benda memiliki implementasi yang sama sekali berbeda dapat memiliki *interface* yang identik.

  Design pattern membantu untuk menentukan *interface* dengan mengidentifikasi elemen kunci dan *type* data yang akan dikirim keseluruh *interface*. *Memento pattern* merupakan contoh yang baik. Ini menggambarkan bagaimana untuk *encapsulate* dan menyimpan di internal state suatu obyek sehingga obyek dapat dikembalikan ke keadaan nanti.

  Design patterns juga menentukan hubungan antara *interface*. Pada keadatan tertentu, obyek sering membutuhkan beberapa kelas untuk memiliki *interface* yang sama, atau dapat menempatkan *interface* dari beberapa kelas. *Decorator* dan *Proxy* memerlukan *interface* untuk *dekorator* dan *proxy* obyek untuk mengidentifikasi *dekorasi* dan *proxy* obyek.

.

4. *UNIFIED MODELING LANGUAGE* (UML)

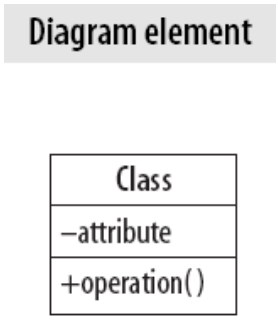
Bagian penting untuk deskripsi *pattern* adalah *Unified Modeling Language* (UML) kelas diagram. UML adalah cara yang diterima secara *universal* menggambarkan *software* dalam bentuk diagram [4]. UML dianggap sebagai standar bahasa pemodelan dengan notasi grafis, dan seperangkat diagram dan elemen. Hal ini digunakan untuk menentukan, memvisualisasikan, memodifikasi, membangun dan mendokumentasikan artefak dari sistem perangkat lunak berorientasi obyek dalam pengembangan [5].

UML didefinisikan sebagai tujuan umum bahasa pemodelan standar di bidang rekayasa perangkat lunak berorientasi obyek. Standar ini dikelola, dan telah dibuat, oleh Management Group Object (OMG). Ini pertama kali ditambahkan ke daftar OMG mengadopsi teknologi pada tahun 1997, dan sejak itu menjadi standar industri untuk sistem perangkat lunak-intensif pemodelan. Ini mencakup seperangkat teknik notasi grafis untuk membuat model visual sistem perangkat lunak-intensif berorientasi obyek [5].

UML adalah alat untuk menentukan dan memvisualisasikan sistem perangkat lunak. Ini termasuk jenis diagram standar yang menggambarkan dan visual memetakan aplikasi komputer atau desain sistem *database* dan struktur. Penggunaan UML sebagai alat untuk menentukan struktur sistem adalah cara yang sangat berguna untuk mengelola sesuatu yang besar, sistem yang kompleks. Memiliki struktur yang terlihat jelas memudahkan untuk memperkenalkan *programmer* baru untuk proyek yang sudah ada [5].

Untuk diagram blok, disajikan sebagai gambar berikut [4] :

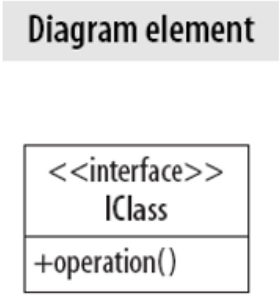
- ***Class***



Gambar 1. Diagram elemen dari *class*

Type dan parameter yang ditentukan sangat penting, akses ditunjukan oleh :
+ menunjukkkan *public*.
- menunjukkkan *private*.
menunjukkkan *proteted*.

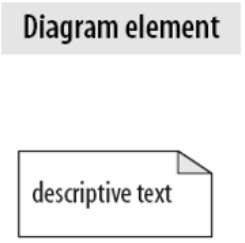
- ***Interface***



Gambar 2. Diagram elemen dari *Interface*

Awalan nama menggunakan I, juga digunakan untuk abstract class.

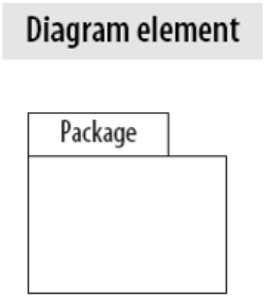
- ***Note***



Gambar 3. Diagram elemen dari *note*

Untuk mendiskripsikan teks.

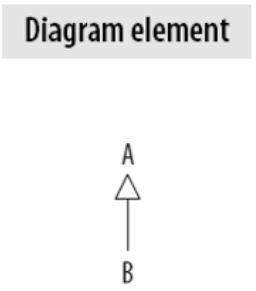
- ***Package***



Gambar 4. Diagram elemen dari *package*

Merupakan *group* dari *class* dan *interface*

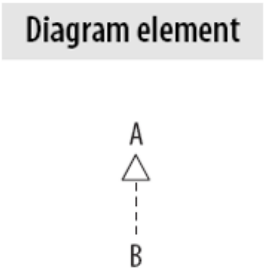
- ***Inheritance***



Gambar 5. Diagram elemen dari *inheritance*

B *inherits* atau mewarisi dari A.

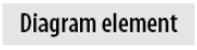
- ***Realization***



Gambar 6. Diagram elemen dari *realization*

B *implements* A

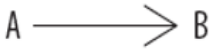
- ***Association***



Gambar 7. Diagram elemen dari *association*

A dan B panggilan dan akses elemen masing-masing.

- ***Association (one way)***



Gambar 8. Diagram elemen dari *association (one way)*

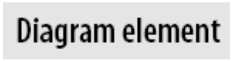
A dapat memanggil dan mengakses elemen B, tetapi tidak sebaliknya

- ***Aggregation***

Gambar 9. Diagram elemen dari *aggregation*

A memiliki B, dan B dapat hidup lebih lama dari pada A.

- ***Composition***



Gambar 10. Diagram elemen dari *composition*

A memiliki B, dan B tergantung pada A.

Terdapat tiga macam blok, untuk *class*, *class interface/abstract* dan *package*. *Class* merupakan elemen diagram utama dan berisi konten yang detail yaitu atribut dan operasi (*method*). Diagram UML tidak memasukkan semua seperti kedalam program, UML berupa unsur-unsur yang penting dalam *pattern*. Akses semua atribut dan operasi sebagai idikasi. Pada dasarnya untuk atribut adalah *private* dan untuk operasi adalah *public*.

5. PERKENALAN BAHASA C# DAN .NET *FREMEWORK*

C # adalah bahasa berorientasi obyek yang elegan dan *type-safe* yang memungkinkan pengembang untuk membangun berbagai aplikasi yang aman dan kuat yang berjalan di .NET Framework. Dapat menggunakan C # untuk membuat aplikasi *Windows client*, layanan XML Web, komponen terdistribusi, aplikasi *client-server*, aplikasi *database*, dan masih banyak lagi. Visual C # menyediakan editor canggih, desainer penggunaan yang mudah, *debugger* terintegrasi, dan *tools* lain untuk membuatnya lebih mudah untuk mengembangkan aplikasi berbasis pada bahasa C # dan .NET Framework [9].

C # 1,0 keluar pada bulan Desember 2002, mewujudkan banyak penelitian di OOP yang telah terjadi sejak Java diluncurkan tujuh tahun sebelumnya. C # 2.0 dirilis dalam bentuk akhir bulan September 2005, dan standar ECMA dibuat tersedia pada bulan Juni 2006. C # 2.0 menambahkan lima fitur yang signifikan untuk C # 1.0 [4].

Sintak C# sangat ekspresif, namun juga sederhana dan mudah dipelajari. Sintaks C# akan langsung dikenali kepada siapa pun akrab dengan C, C ++ atau Java. *Developer* yang tahu bahasa ini biasanya dapat mulai bekerja secara produktif di C# dalam waktu yang sangat singkat. C# sintaks menyederhanakan banyak kompleksitas C ++ dan menyediakan fitur canggih seperti *nullable value types, enumerations, delegates, lambda expressions* dan akses memori langsung, yang tidak ditemukan di Java. C # mendukung *generic methods* dan *types*, yang memberikan peningkatan keamana *type* dan *performance*. *Language-Integrated Query* (LINQ) ekspresi membuat query yang praktis [9].

Sebagai bahasa berorientasi obyek, C # mendukung konsep *encapsulation, inheritance*, dan *polymorphism*. Semua *variabel* dan *method*, termasuk main *method*, *entry point* aplikasi, yang dibungkus dalam definisi kelas. Sebuah kelas dapat mewarisi langsung dari satu kelas induk, dapat mengimplementasikan sejumlah *interface*. *Method* yang menimpa *method virtual* dalam kelas induk memerlukan kata kunci *override* sebagai cara untuk menghindari kecelakaan redefinisi. Dalam C #, *struct* seperti kelas *lightweight* adalah jenis tumpukan yang dapat mengimplementasikan *interface* tetapi tidak mendukung warisan [9].

Selain prinsip-prinsip berorientasi obyek dasar, C# membuatnya mudah untuk mengembangkan komponen perangkat lunak melalui beberapa bahasa konstruksi yang inovatif, termasuk yang berikut:

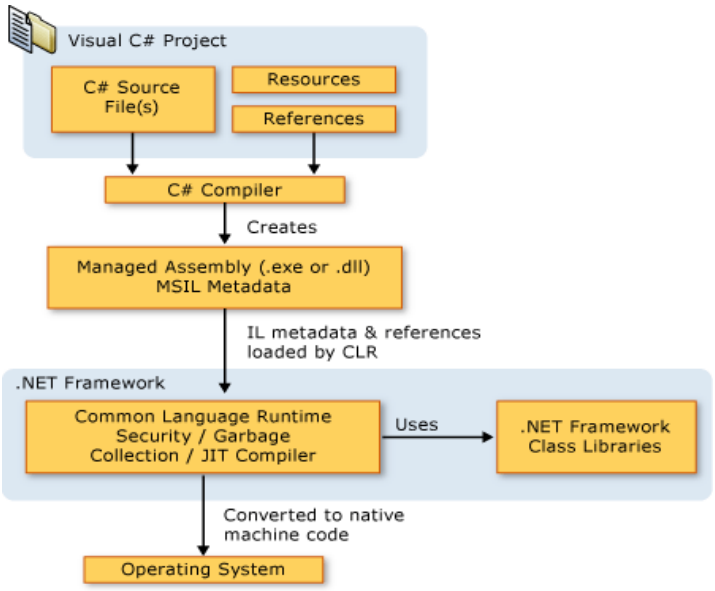
- *Encapsulated* method disebut dengan delegate, merupakan cara aman untuk notification.
- Properti, yang berfungsi sebagai *private* member *variables*.
- Atribut yang menyediakan metadata deklaratif tentang *types* ketika *run* time.
- *Inline* dokumentasi XML
- LINQ yang menyediakan kemampuan *query built-in* di berbagai *sources* data.

Jika harus berinteraksi dengan *software Windows* lainnya seperti COM atau Win32 DLL, C# mampu melakukan ini melalui proses yang disebut *Interop*. *Interop* memungkinkan C# untuk melakukan apa yang dapat dilakuan oleh C ++. C# bahkan mendukung *pointer* dan konsep *unsafe* untuk kasus-kasus di mana akses memori langsung harus dilakukan [9]..

C# membuat proses menjadi sederhana dibandingkan dengan C dan C ++ dan lebih fleksibel daripada Java. Tidak ada file *header* yang terpisah, dan tidak ada persyaratan bahwa *method* dan *type* dinyatakan dalam urutan tertentu. C# *source file* dapat menentukan sejumlah *classes, structs, interfaces, and events* [9].

C# program berjalan di .NET *Framework*, komponen *integral* dari *Windows* yang mencakup sistem eksekusi virtual yang disebut *common language runtime* (CLR) dan *class library*. CLR merupakan implementasi komersial oleh microsoft *common language infrastructure* (CLI), sebuah standar internasional yang merupakan dasar untuk menciptakan *execution* dan *development environments* di mana *languages* dan *libraries* bekerja sama dengan baik. *Source code* yang ditulis dalam C # *compiled* menjadi *intermediate language* (IL) yang sesuai dengan spesifikasi CLI. IL code dan resources, seperti bitmap dan string, disimpan pada *disk* dalam sebuah file eksekusi bernama *assembly*, biasanya dengan ekstensi .exe atau .dll.

Ketika program C# dijalankan, *assembly* dimuat ke dalam CLR, yang mungkin mengambil berbagai tindakan berdasarkan informasi dalam manifes. Kemudian, jika persyaratan keamanan terpenuhi, CLR *melakukan just in time* (JIT) kompilasi untuk mengkonversi kode IL ke instruksi mesin. CLR juga menyediakan layanan lain yang berhubungan dengan *automatic garbage collection*, *exception handling*, dan manajemen *resource*. Kode yang dieksekusi oleh CLR ini kadang-kadang disebut sebagai *managed code*, berbeda dengan *unmanaged code* yang *compiled* ke dalam bahasa mesin yang menargetkan sistem tertentu. Diagram berikut menggambarkan saat *compiled* dan *run-time* hubungan C# *source code files*, .NET *Framework class libraries*, *assemblies*, dan CLR [9].



Gambar 10. Proses *compiled dan run-time*

Language interoperability adalah fitur dari .NET Framework. Karena kode IL yang dihasilkan oleh C # compiler sesuai dengan *Common Type Specification* (CTS), kode IL yang dihasilkan dari C # dapat berinteraksi dengan kode yang dihasilkan dari versi NET Visual Basic, Visual C ++, atau lebih dari 20 lainnya bahasa CTS-compliant. Satu *assembly* dapat berisi beberapa modul yang ditulis dalam bahasa NET yang berbeda, dan *type* dapat mereferensi satu sama lain, seperti jika ditulis dalam bahasa yang sama [9].

6. *ROLE PROXY PATTERN*

Proxy pattern mendukung obyek untuk mengontrol dalam membuat dan mengakses objek lain. *Proxy* biasanya sebuah obyek yang kecil yang teridiri dari obyek yang komplek yg bersifat private, yang diaktifkan setelah terjadi suatu dalam keadan tertentu [4].

7. ILUSTRASI *PROXY PATTERN*

Facebook merukan salah satu jaringan media sosial yang populer sejak tahun 2004 sampai sekarang [8].

Gambar 11. Ilustrasi *proxy pattern*, halama *facebook* setelah *login*

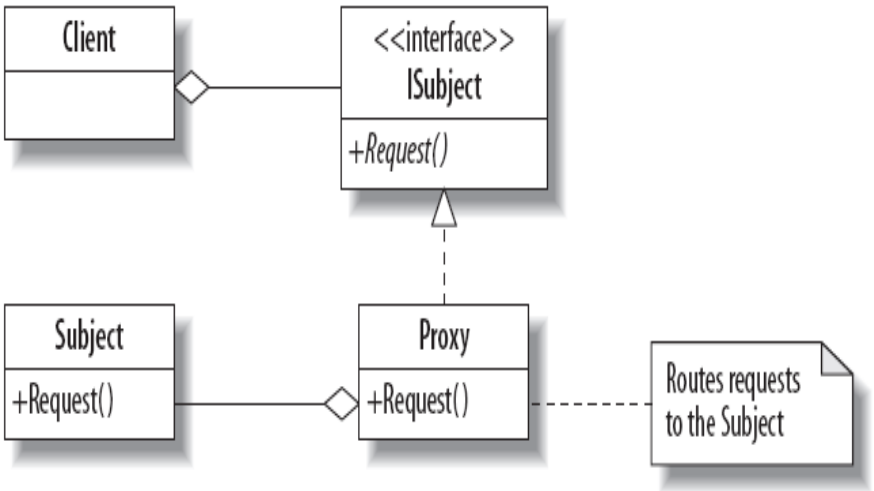
Gambar 12. Ilustrasi *proxy pattern*, halaman *facebook* sebelum *login*

Dari ilustrasi gambar 11 dan gambar 12 terdapat perbedaan dan persaman. Kedua gambar mengakses halaman sama yaitu *facebook.com* namun memiliki perbedaan tampilan. Untuk seperti tampil pada seperti gambar 11, maka harus mendaftar terlebih dahulu sebagai *user*. Formulir pendaftaran terdapat pada gambar 12. Setelah mendaftar masih belum tampil seperti gambar 11, *user* harus menambahkan teman, pemainan dan beberapa fitur facebook yang diinginkan.

Setelah *user* mempunyai *user name* dan *password*, *user* langsung dapat *login*. Ketika masuk halaman, user dapat melihat aktifitas teman yang sudah di tambahkan. Selain itu beberapa fitur lain adalah mengikuti seseorang, grop, bahkan mengakses kehalaman teman, sehingga bisa melihat aktifitas teman. Kalau dipahami terdapat beberapa obyek yang terdapat pada halaman *facebook*, yang dapat diakses sebelum dan setelah login. Ilustrasi ini merupakan mekanisme dari *proxy pattern* yaitu ketika *regristrasi* dan *login*.

8. DESAIN UML *PROXY PATTERN*

Desain dari *proxy pattern* dan elemen-elemenya ditampilkan pada gambar 13.



Gambar 13. Diagram UML *proxy pettern*

Berikut merupakan elemen-elemen dari *proxy pettern* :

- *ISubject*
  Interface yang akan digunakan oleh *Client*.
- *Subject*
  Class yang mempresentasikan sebagai *proxy*.
- *Proxy*
  Class yang dibuat, mengontol *enhances*, dan *authenticates access* ke *Sebject*.
- *Request*
  Operasi pada *Subject* yang diarahkan melalui *proxy*.

Pusat *class* yaitu *proxy*, dimana mengimplementasi *interface Isubject*. *Clien* dapat menggunakan *Isubject* untuk membuat obyek dari *proxy*. Sebenarnya dalam implementasi *proxy pattern* tidak harus menggunkan *interface*, dapat menggunakan *class* biasa. *Proxy* melakukan tugasnya di *frontend*. Dengan membuat *Subject* kelas menjadi *private*, menjadikan class *Subject* tidak dapat di akses kecuali melalui *Proxy* [4].

Berikut merupakan perbandingan antara elemen-elemen *proxy pattern* dan *facebook*[4].

`ISubject`	A list of actions that can be done on the pages
`Subject`	Actual pages belonging to one person
`Subject.Request`	Actually changing the pages
`Proxy`	Frontend to the actual pages
`Proxy.Request`	Performs some action before routing the `Request` onward
`Client`	A user visiting her pages

Gambar 14. Persaman UML *proxy pattern* dengan *facebook*

Ada beberapa jenis *proxy*, yang paling umum di antaranya adalah [4]:

- *Virtual proxies*
  Menciptakan obyek ke obyek lain, ini berarti objek tidak akan dibuat sampai benar-benar diperlukan. Berfungsi Jika proses pembuatan objek sangat lambat misalnya terlalu banyak memakan *memory* atau obyek terbukti tidak perlu.
- *Authentication proxies*
  Bertindak sebagai persetujuan untuk mengakses, seandainya kondisi *request* benar. Hal ini menjadikan *proxy* sebagai keamanan, karena membutuhkan *Authentication* sebagai persetujuan untuk mengakses sesuatu.
- *Remote proxies*
  Untuk meminta dan mengirimkan ke lain jaringan. Hal ini mirip dengan konsep penggunaan layanan web atau layanan WCF di mana klien harus mengakses *remote resource* pada jaringan lain.
- *Smart proxies*
  Menambahkan atau merubah *request* sebelum dikirim. Digunakan untuk menambahkan beberapa jenis fungsi untuk mengelola resources secara efisien.

Pada contoh ilustrasi *proxy* yaitu *facebook*, dapat diuraikan sesuai jenis *proxy* yaitu sebagai berikut:

- *Virtual proxies*
  Menunda dalam pembuatan obyek yang belum dibutuhkan, misalkan melihat pada halaman utama *facebook*. Pada halaman utama ketika belum login, maka tidak perlu untuk membuat obyek yg berisi *timeline*. Selain tidak berguna, ini dapat memakan *memory*.
- *Authentication proxies*
  Ini sangat terlihat jelas ketika login *facebook*. Ketika *user name* dan *password* benar, maka akan dapat mengakses obyek-obyek yang sebelumnya tidak dapat di akses.
- *Remote proxies*
  Mengirimkan *request* ke jaringan lain.

- *Smart proxies*
  Aksi performa pada *friend's books.*

9. IMPLEMENTASI

Secara fisik, program C # terdiri dari satu atau lebih unit kompilasi, masing-masing terdapat dalam *file source* terpisah. Ketika program C # dikompilasi, semua unit kompilasi diproses bersama-sama. Dengan demikian, unit kompilasi dapat bergantung satu sama lain, mungkin dalam mode melingkar. *assembly* merupakan *physical container* untuk *types* dan *resources* yang terkait dengan mereka [4].

Pada tingkat logis, C # program yang terorganisir menggunakan *namespaces*. *Namespaces* yang digunakan sebagai cara untuk menampilkan unsur-unsur program ke program lain, baik sama atau beda assemblies. *Access modifiers* merupakan cara yang digunakan untuk menetapkan *member* dari *namespace* yang dapat diakses atau tidak [4].

Di dalam C#, *access modifier* mengubah *default* dari *type* atau *member* elemen. Berikut merupakan penjelasan tentang *access modifier* [6]:

- *Private*
  Private access merupakan akses yang paling permisif. Private hanya dapat diakses dalam kelas itu sendiri atau struct di mana mereka di inisialisasi. seperti dalam contoh ini :

```
class Employee
{
    private int i;
    double d;   // private access by default
}
```

  Secara *default field, property, method* memiliki akses *private*. Variabel i dan d hanya dapat diakses dalam *class Employee*. Seandainya *class Employee* dijadikan obyek, maka variabel i dan d tidak dapat diakses.

- *Internal*
  Internal types hanya dapat diakses dalam *assembly* yang sama.

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

- *Protected*
  Protected dapat diakses oleh *class* sendiri dan *class* turunannya. Berikut merupakan contoh implentasi *protected* yang dapat diakses *class* turunannya.

```
class A
{
  protected int x = 123;
}

class B : A
{
  static void Main()
  {
    A a = new A();
    B b = new B();

    // Error CS1540, because x can only be accessed by
    // classes derived from A.
    // a.x = 10;

    // OK, because this class derives from A.
    b.x = 10;
  }
}
```

- *Public*
  Public dapat diakses disemua level. Perlu hati-hati dalam menggunakannya. Berikut merupakan contoh dari *Public*:

```
class PointTest
{
   public int x;
   public int y;
}

class MainClass4
{
   static void Main()
   {
      PointTest p = new PointTest();
      // Direct access to public members:
      p.x = 10;
      p.y = 15;
      Console.WriteLine("x = {0}, y = {1}", p.x, p.y);
   }
}
// Output: x = 10, y = 15
```

Pada dasarnya sebagai pembatas hak akses suatu *class, property, field, structs, interfaces, enums, delegates* maupun method.

Berdasar menguraian teori *proxy*, maka dapat dijelaskan sebagi berikut [4]:

- Bagaimana membagi antara Client pada sisi yang satu, proxy, dan subject sebagai sisi yang lain untuk dapat mengimplemtasikan access modifiers.
- Bagaimana virtual dan protection proxies melakukan tugasnya dan rutenya.

Berikut merupakan inti dari virtual proxy [4]:

```
public class Proxy : Isubject
{
   Subject subject;
   public string Request()
   {
      // A virtual proxy creates the object only on its first method call
      if (subject == null)
         subject = new Subject();
      return subject.Request();
   }
}
```

Clien dapat menciptakan Proxy setiap saat dengan menggunakan normal instantiation. Karena konstrakor yg digunakan secara default, reference Subject tidak dibuat secara langsung. Hanya ketika method Request di panggil, Subject reference akan di cek, jika masih null maka obyek baru akan dibuat.

Berikut merupakan implementasi dari proxy pattern secara sederhana, ini menerapkan sesuai dengan teori proxy yang dijelaskan diatas [4].

```
1 using System;
2
3
4 // Shows virtual and protection proxies
5
6 class SubjectAccessor {
7 public interface ISubject {
8 string Request ( );
9 }
10
11 private class Subject {
12 public string Request( ) {
13 return "Subject Request " + "Choose left door\n";
14 }
15 }
16
17 public class Proxy : ISubject {
```

```
18 Subject subject;
19
20 public string Request( ) {
21 // A virtual proxy creates the object only on its first method call
22 if (subject == null) {
23 Console.WriteLine("Subject inactive");
24 subject = new Subject( );
25 }
26 Console.WriteLine("Subject active");
27 return "Proxy: Call to " + subject.Request( );
28 }
29 }
30
31 public class ProtectionProxy : ISubject {
32 // An authentication proxy first asks for a password
33 Subject subject;
34 string password = "Abracadabra";
35
36 public string Authenticate (string supplied) {
37 if (supplied!=password)
38 return "Protection Proxy: No access";
39 else
40 subject = new Subject( );
41 return "Protection Proxy: Authenticated";
42 }
43
44 public string Request( ) {
45 if (subject==null)
46 return "Protection Proxy: Authenticate first";
47 else return "Protection Proxy: Call to "+
48 subject.Request( );
49 }
50 }
51 }
52
53 class Client : SubjectAccessor {
54 static void Main( ) {
55 Console.WriteLine("Proxy Pattern\n");
56
57 ISubject subject = new Proxy( );
58 Console.WriteLine(subject.Request( ));
59 Console.WriteLine(subject.Request( ));
60
61 ProtectionProxy subject = new ProtectionProxy( );
62 Console.WriteLine(subject.Request( ));
63 Console.WriteLine((subject as ProtectionProxy).Authenticate("Secret"));
64 Console.WriteLine((subject as ProtectionProxy).Authenticate("Abracadabra"));
65 Console.WriteLine(subject.Request( ));
66 }
67 }
68
69 /* Output
70
71 Proxy Pattern
72
73 Subject inactive
74 Subject active
75 Proxy: Call to Subject Request Choose left door
76
```

```
77 Subject active
78 Proxy: Call to Subject Request Choose left door
79
80 Protection Proxy: Authenticate first
81 Protection Proxy: No access
82 Protection Proxy: Authenticated
83 Protection Proxy: Call to Subject Request Choose left door
84 */
```

Untuk memudahkan menjelasan program, penjelasan akan dilakukan dari sisi *Client*. Program akan mulai dijalankan pada *method main*. Pada baris ke 57, program akan membuat obyek dari kelas *proxy* dengan memberinama *subject* terletak di *client*. Dalam membuat obyek *subject* tidak terjadi apa-apa, seperti dijelaskan sebelumnya.yaitu didalam *proxy class* konstraktor secara *default*. Pada baris ke 52, ini menerapakan *virtual proxy*. Pada bari 58 method *Request* akan di panggil. Pertama kali obyek dari *classs Subject* akan di cek, apakah sudah dibuat atau belum terdapat pada baris 22. Karena pertama kali, maka obyek masih *null*. Untuk itu baris 24 akan dijalankan. Pada baris 24 berfungsi untuk membuat obyek *subject* dalam kelas proxy. Pada baris 59, akan memanggil *method Request* lagi, disini *object subject* sudah tidak *null*.

Pada baris 61, terjadi perubahan jalur. Proses perpindahan jalur dengan menimpa *object subject* yang terdapat pada *client*. Obyek *subject* yang baru berasal dari *class ProtectionProxy,* dimana *class ProtectionProxy* juga imlementasi dari *ISubject*. Sama seperti penjelasan sebelumnya, pada baris 62 akan memanggil *method Request*. Didalam *method Request* terdapat pengujian apakah obyek *subject* sudah dibuat. Karena obyek *subject* masih bernilai null, maka disarankan untuk memanggil *method Authenticate*.

Pada *method Authenticate* membutuhkan parameter *string*, ini digunakan sebagai *password* dalam pembuatan obyek *subject* yang berada didalam *class proxy*. Pada baris 63, parameter yang dimasukkan adalah "*Secret*", sedangkan *password* yang telah di-*hardcode* adalah *Abracadabra*. Maka pada baris 37, sebagai fungsi pengecekan akan memenuhi kreteria yaitu *password* tidak cocok.

Proses autentikasi diulangi lagi pada baris 64, dengan parameter *Abracadabra*. Pada proses baris 37 tidak memenuhi syarat, sehingga baris 40 dijalankan yaitu pembuatan obyek *subject* dari *class Subject*. *Class Subject* merupakan *class* yg dilindungi, *class* ini tidak dapat dipanggil sebelum proses autentifikasi dengan *password* benar.

Pada baris 65, *method Request* yang berada di *class proxy* dipanggil lagi, kali ini obyek *subject* sudah tidak null, karena proses autentifikasi sudah benar. Sehingga *method Request* yang berada didalam obyek *subject* dapat dipanggil.

10. PERBANDINGAN ILUSTRASI DENGAN CONTOH IMPLEMENTASI

Pada ilustrasi diatas menggunakan *facebook* untuk memudahkan pembahasan. Pada *facebook* menggunakan *virtual proxies* dan *authentication proxies,* berikut merupakan penjelasan perbandingan antara ilustrasi dengan contoh implementasi :

- Class Subject merupakan data yang dilindungi, pada facebook seperti *timeline*.
- Menggunakan password pada proses autentifikasi, sehingga dapat mengakses data yang dilindungi. Pada *facebook* perlu melakukan login dengan memasukan *user name* dan *password* untuk bisa mengakses *timeline*.
- Sebelum proses autentifikasi berhasil, obyek *Subject* tidak akan dibuat. Pada *facebook* sebelum login berhasil, obyek *timeline* tidak akan ditampilkan. Ini fungsi *virtual proxies* sebagai menghemat *memory*.

11. KESIMPULAN

Tujuan dari *design pattern* adalah membuat kode program menjadi mudah, bersih dan aman. Dengan adanya *design pattern* membuat lebih mudah dalam menyelesaikan masalah, karena sudah ada bentuk arsitektur dalam menyelesaikan masalah. Misalnya dalam membuat *login*, maka dapat menggunakan arsitektur dari *proxy pattern*. Dengan *pattern* juga membuat kode yang dibuat menjadi bersih, terlihat dari pembagian elemen sesuai fungsi untuk menghindari dari kode yang berantakan.

Proxy pattern merupakan salah satu jenis *design pattern* yang berfungsi sebagai keamanan *software*. *Proxy pattern* bertindak sebagai gerbang autentifikasi sebelum mengakses data yang bersifat rahasia. Diilustrasikan dengan halaman *facebook* yang harus *login* terlebih dahulu sebelum melihat *timeline*. Selai itu dapat menghemat memory, karena tidak perlu membuat obyek *timeline* sebelum *login* berhasil.

REFERENSI

[1] E. E. OGHENEOVO, P. O. ASAGBA,"A Comparative Analysis of Structured and Object-Oriented Programming Methods," JASEM ISSN 1119-8362, Vol. 12, no. 4, pp. 41-46, Desember 2008.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Boston: Addison-Wesley, 1995.

[3] G. Fischer, "The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design," Proceedings of ISFST2001: International Symposium on Future Software Technology, Software Engineers Association, Japan, pp. 1-8, November 2001.

[4] J. Bishop, C# 3.0 Design Patterns, Edisi Pertam, 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Desember2007.

[5] S. Lee, "Unified Modeling Language (UML) for Database Systems and Computer Applications," International Journal of Database Theory and Application, Vol. 5, No. 1, March 2012.

[6] ------, "Access Modifiers (C# Reference)," 2016. [online]. Tersedia: https://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx [Diakses: 1 Juni 2016]

[7] ------, "Design pattern," modified on 18 November 2015. [online]. Tersedia: https://en.wikipedia.org/wiki/Design_pattern [Diakses: 10 Mei 2016]

[8] ------, "Facebook," diperiksa pada tanggal 16 Mei 2016. [online]. Tersedia: https://id.wikipedia.org/wiki/Facebook [Diakses: 1 Juni 2016]

[9] ------, "Introduction to the C# Language and the .NET Framework," 2016. [online]. Tersedia: https://msdn.microsoft.com/en-us/library/z1zx9t92.aspx [Diakses: 11 Mei 2016]