# BAB III

# ANALISIS DAN PERANCANGAN APLIKASI

## 3.1 Metode Penelitian

Hal-hal yang dilakukan dalam penelitian ini antara lain

1. Studi Literatur

   Pada tahap ini akan dilakukan studi dari buku, jurnal dan artikel ilmiah mengenai teori pembuatan *Call Graph* dan algoritma XTA, yang akan di-implementasikan. Selain itu, juga dilakukan pembelajaran mengenai dasar-dasar dari pembuatan aplikasi (*plug-in*), seperti tata cara pembuatan *plug-in* Eclipse dan penggunaan AST (*Abstract Syntax Tree*).

2. Desain Aplikasi

   Pada tahap ini dilakukan perancangan dasar dari alur jalannya aplikasi, dengan menentukan input-output aplikasi dan membagi-bagi proses menjadi beberapa sub-proses secara umum. Selain itu juga dilakukan perancangan antarmuka dari aplikasi.

3. Pembuatan Aplikasi

   Tahap ini dilakukan pembangunan dari aplikasi (*plug-in*). Dalam proses pengembangannya digunakan metode prototyping.

4. Uji Coba dan Evaluasi

   Pada tahap ini, akan dilakukan uji coba untuk mengetahui apakah keluaran / *output* dari aplikasi yang dibuat, sudah sesuai dengan tujuan penetitian yang telah dinyatakan. Parameter yang diuji dalam penelitian adalah

jumlah *node*, *edge*, waktu yang diperlukan dan memori yang terpakai. Selain itu juga dilakukan evaluasi untuk mengetahui respon dari pengguna terhadap aplikasi *plug in* yang dibuat.

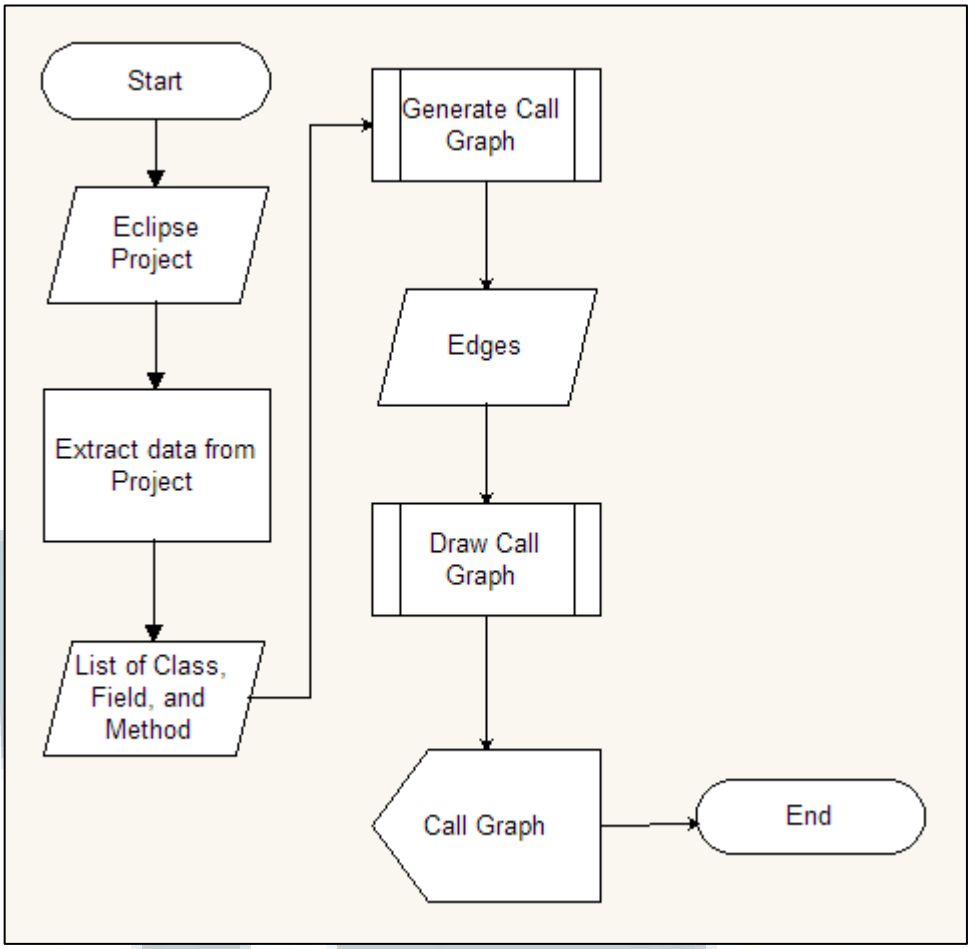
5. Penulisan Skripsi

   Pada tahap ini, dilakukan penulisan skripsi yang merupakan dokumentasi dari penelitian yang dilakukan.

## 3.2 Spesifikasi Umum Aplikasi

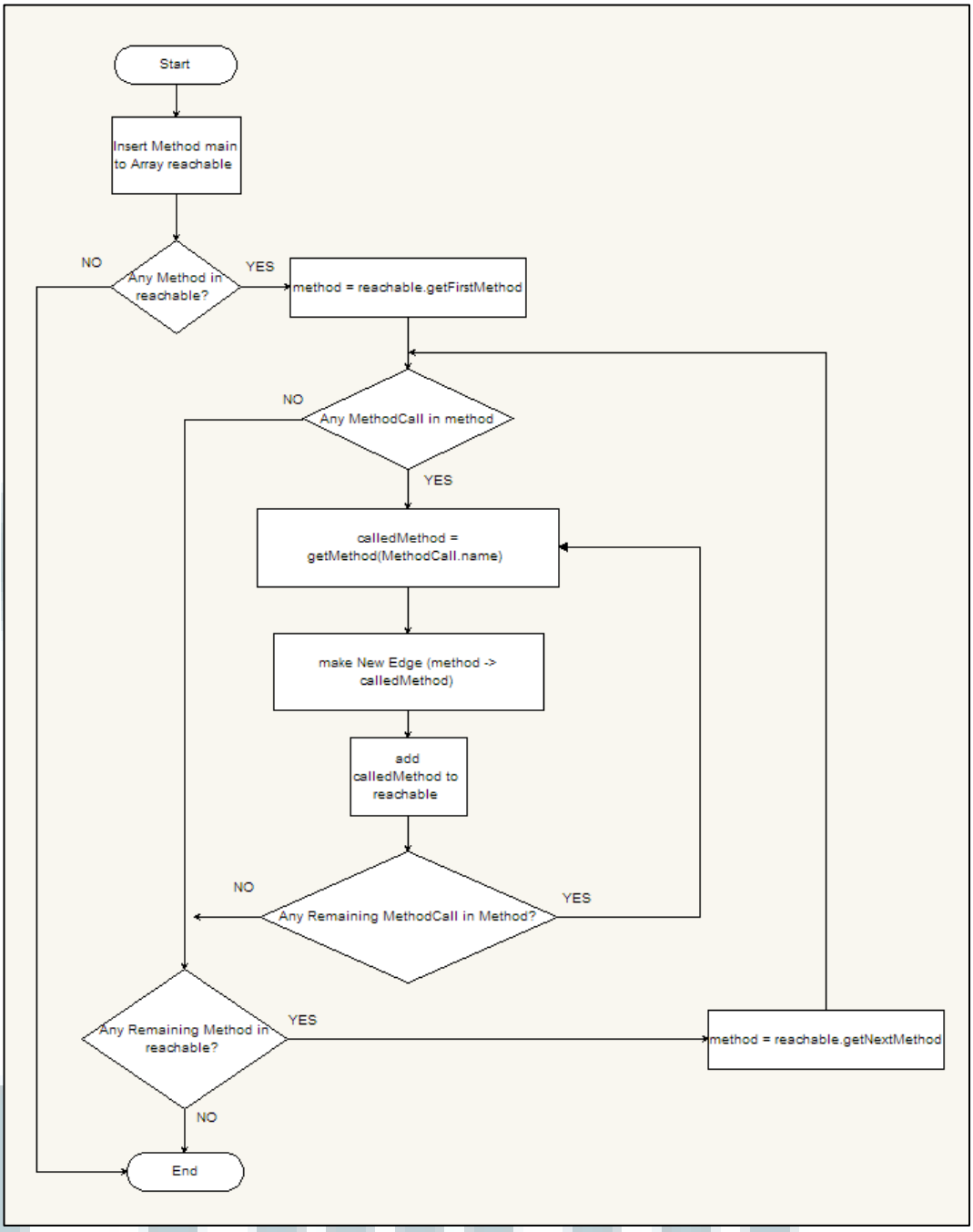
Tujuan dari pembuatan aplikasi ini adalah untuk menghasilkan keluaran berupa *Call Graph* dari *source code* berbahasa JAVA. Langkah-langkah dari aplikasi yang dibuat dapat dilihat dalam *flowchart* berikut.

Berdasarkan Gambar 3.1, aplikasi dimulai dengan mendeteksi *Workspace* yang sedang dikerjakan, mencari tahu nama *project* dan *file-file* yang digunakan dalam *project*. Kemudian setelah user memerintahkan aplikasi untuk memulai proses *generate*, proses ekstrak *method* pun dimulai. Pada proses ini, informasi tentang *class* dari *source code* akan diambil. Untuk membuat *source code* yang akan di ekstrak menjadi lebih terstruktur, digunakan AST (*Abstract Syntax Tree*) untuk membantu proses ini. Dari source code yang ada, akan dibuat AST-nya dan dari setiap *node* AST, bisa didapat berbagai data informasi yang dibutuhkan yaitu nama *class*, *field*, *method*, beserta *parameter*, *return type* dan beberapa informasi yang dibutuhkan pada bagian *body*.




Gambar 3.1 *Flowchart* Umum Aplikasi pembentukan Call Graph.

Informasi–informasi yang didapat pada proses *extract method* akan menjadi modal utama pada proses selanjutnya, yaitu proses *generate call graph*. Karena itu, aplikasi ini mengasumsikan bahwa *source code* sudah dalam keadaan benar secara *syntax* atau bisa di-*compile*, sehingga bisa menghasilkan *output* yang benar.




Gambar 3.2 *Flowchart* Proses Generate Call Graph (Reachability Analysis).

Dalam penelitian ini, selain menggunakan XTA sebagai algoritma pembentukan Call Graph, juga akan digunakan algoritma *Reachability Analysis* (algoritma pembentukan call graph yang paling sederhana), sebagai pengukur ke-

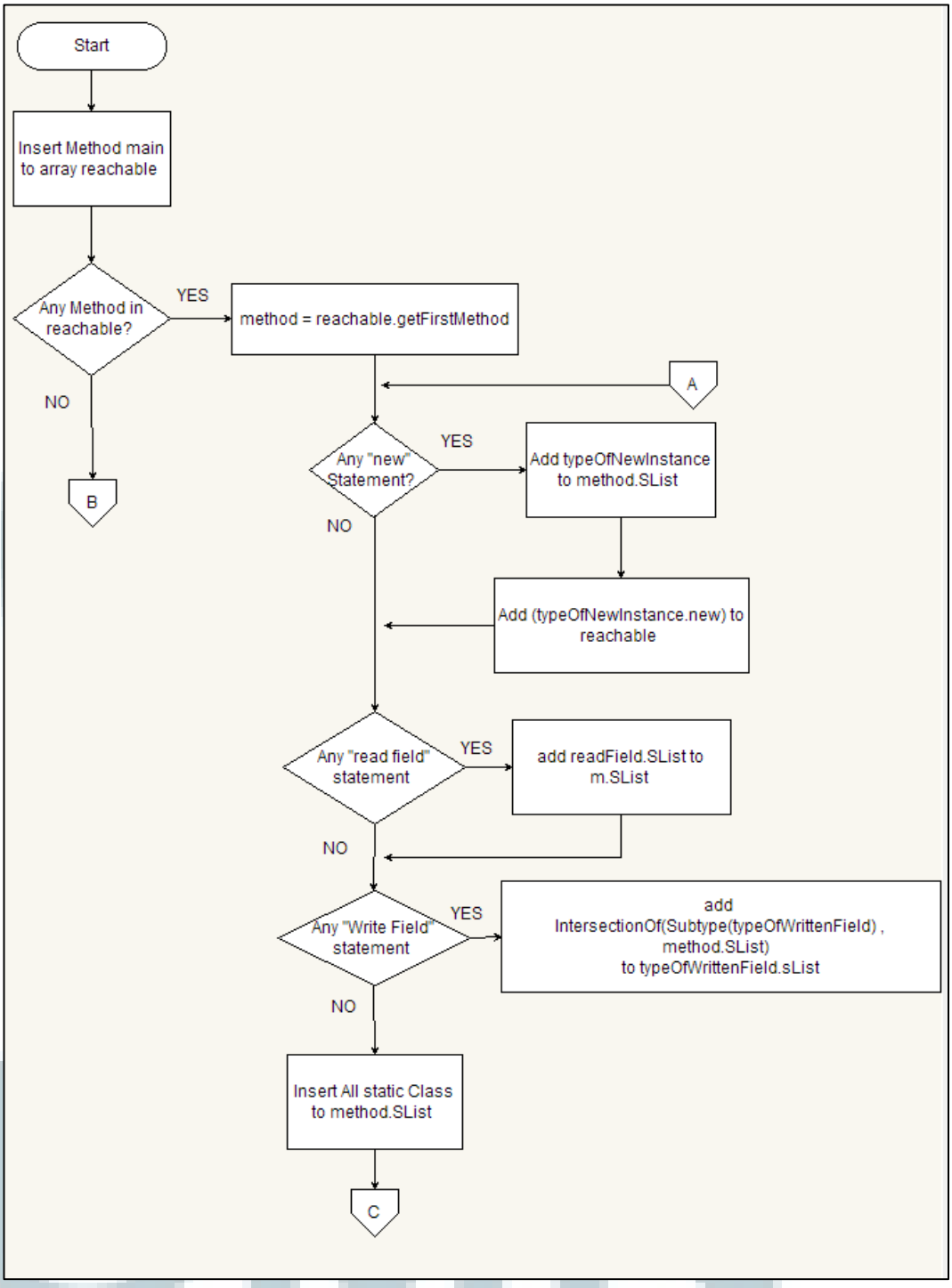
efektifitas-an algoritma XTA. Algoritma RA dimulai dengan, memasukkan *method* "main" (*entry point* dari JAVA *Project*) kedalam sebuah *list reachable*. Setelah itu, untuk setiap *method* pada *reachable* akan di cek apakah ada *statement* "pemanggilan method" pada *method* tersebut. Jika ada, semua *method* yang memiliki nama yang sama dengan nama method yang dipanggil (pada setiap *class*), akan ditambahkan kedalam *reachable* dan terbentuk *edge* [method -> methodDipanggil]. Proses – proses tersebut akan diulang sampai seluruh *method* dalam *reachable* sudah habis diproses.

XTA membutuhkan variabel (SList) untuk setiap *method* dan *field*. Algoritma ini dimulai dengan memasukkan method main (*entry point*) kedalam *array reachable*. Dengan berdasarkan *flowchart* yang ada pada Gambar 3.3 dan 3.4, secara umum, cara kerja algoritma XTA ini memiliki 4 kondisi utama, yaitu

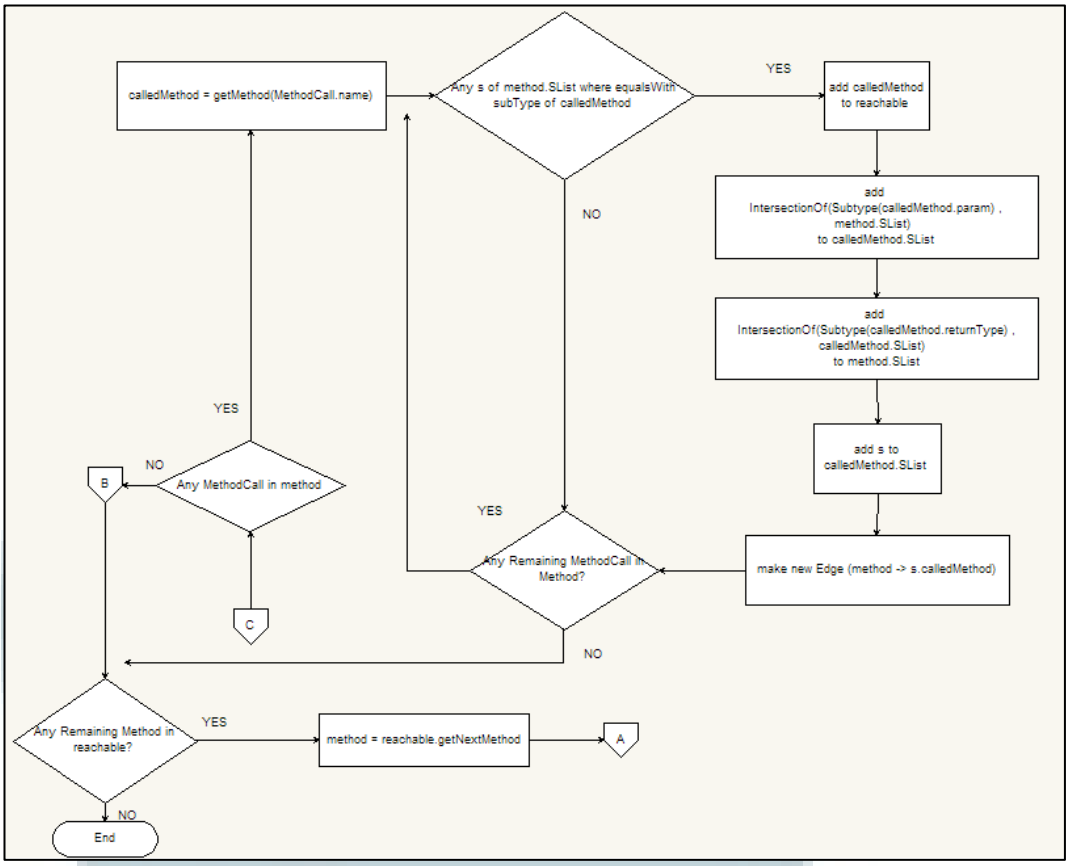
1. Jika terdapat *statement* "new" pada *body method*, tipe dari *instance* yang di-*new* tersebut akan dimasukkan kedalam SList dari *method* dan juga dimasukkan kedalam *reachable*.
2. Jika terdapat *statement* pembacaan *field* pada *body method*, seluruh SList dari *field* yang dibaca tersebut, akan di masukkan kedalam SList milik *method*.
3. Jika terdapat statement penulisan *field* pada *body method*, irisan dari *sub-type field* yang ditulis dengan SList *method*, akan dimasukkan kedalam SList milik *field* yang ditulis tersebut.

4. Jika terdapat pemanggilan *method*, akan dicek apakah *sub-type* dari *class method* yang dipanggil berada dalam daftar SList milik *method*. Jika ada, maka :
   - Tambahkan *method* yang dipanggil kedalam *reachable*
   - Irisan dari *sub-type parameter method* yang dipanggil, dengan SList *method*, dimasukkan kedalam SList milik *method* yang dipanggil
   - Irisan dari *sub-type return-type* dari *method* yang dipanggil, dengan SList *method* yang dipanggil, dimasukkan kedalam SList *method*.
   - Tambahkan class dari *method* yang dipanggil kedalam SList milik *method* yang dipanggil
   - Tambahkan *edge* [method -> s.(method yang dipanggil)]

Proses – proses tersebut akan diulang sampai seluruh method dalam *reachable* sudah habis diproses.

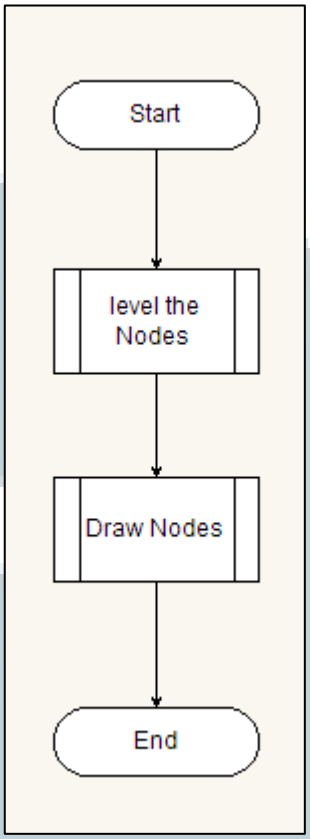



Gambar 3.3 *Flowchart* Proses Generate Call Graph (XTA).



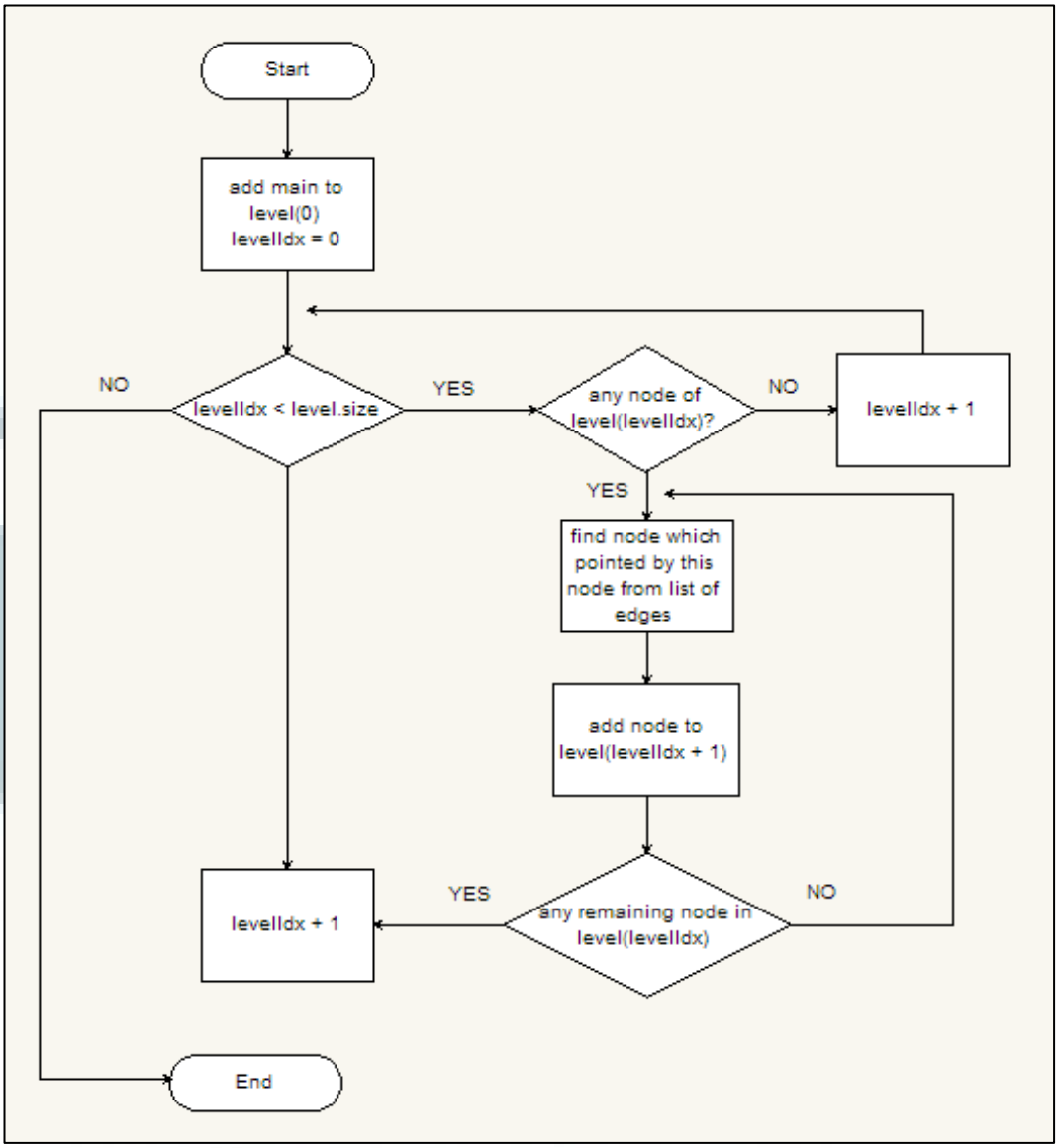

Gambar 3.4 *Flowchart* Proses Generate Call Graph (XTA) (lanjutan).

Setelah proses *generate call graph* selesai, *edge* yang dihasilkan pada proses tersebut akan di visualisasikan dalam bentuk grafik. Proses ini dinamakan sebagai proses *Draw Call Graph*.



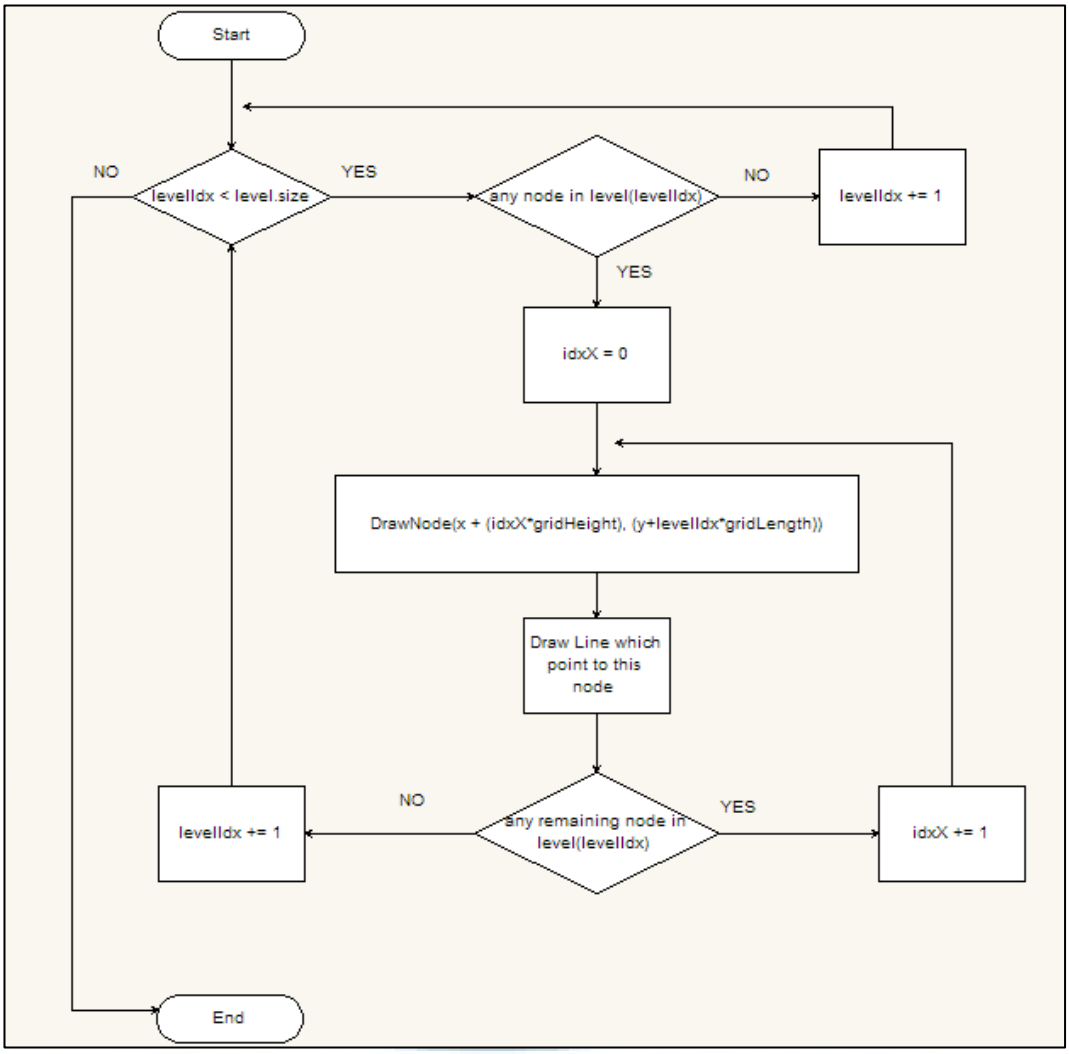

Gambar 3.5 *Flowchart* Proses *Draw Call Graph.*

Proses ini dimulai dengan meletakkan *node-node* yang ada, pada *level* yang sesuai, berdasakan edge yang didapat dari proses *generate call graph*. Proses ini bertujuan untuk memposisikan *node* pada tempat yang sesuai (menyerupai *tree*), sehingga *call graph* bisa tampak lebih rapi.




Gambar 3.6 *Flowchart sub-proses* Level Node.

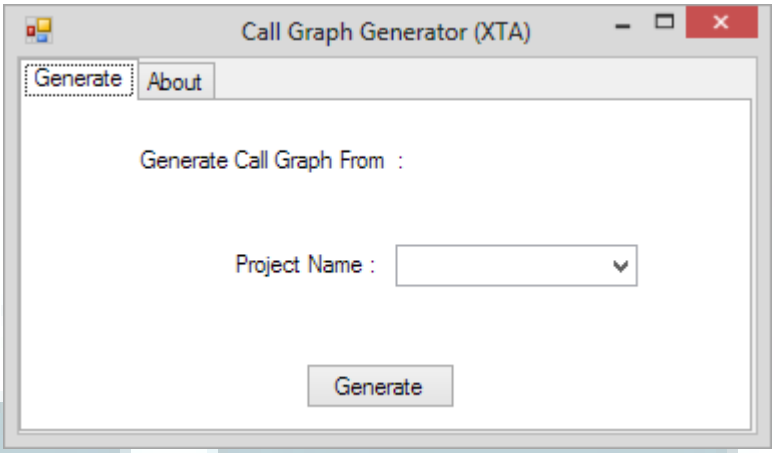
Setelah *node* dimasukkan per-*level*, proses dilanjutkan dengan proses gambar, yaitu meletakkan *node* pada posisi yang sesuai dan juga menggambarkan garis panah antar *node* yang terhubung berdasarkan daftar *edge*.




Gambar 3.7 *Flowchart* sub-proses Draw Node.

### 3.3 Desain Antarmuka Aplikasi

Berikut adalah desain antarmuka dari aplikasi Call Graph Generator (XTA). Desain antarmuka di rancang menggunakan Microsoft Visual Studio 2008, untuk memberikan gambaran yang jelas akan komponen-komponen yang ada.

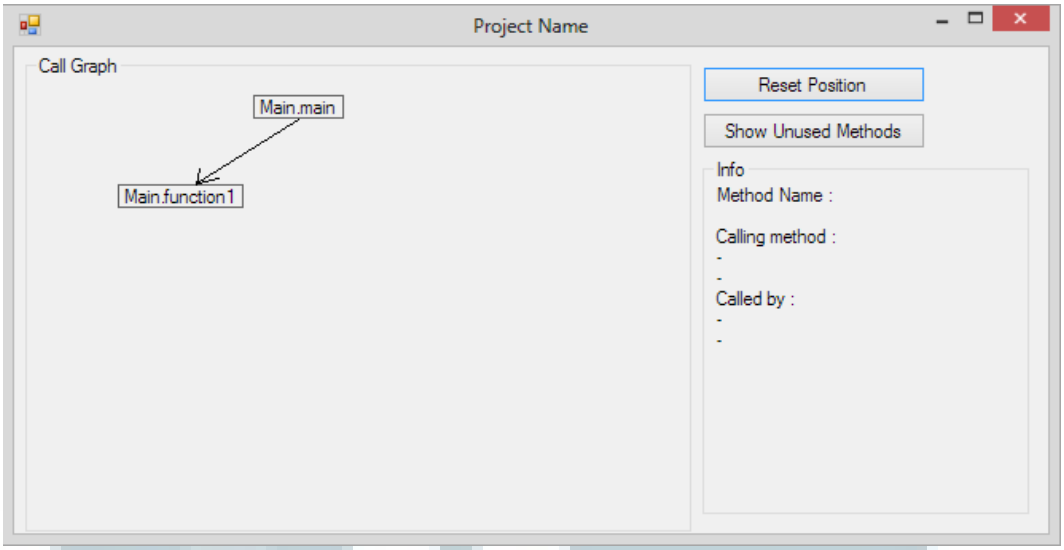



Gambar 3.8 Desain antarmuka menu utama.

Bahasa yang akan digunakan dalam aplikasi adalah bahasa Inggris, dengan alasan bahwa aplikasi ini bisa digunakan oleh semua programmer Eclipse, tidak terbatas oleh negara. Maka dari itu, bahasa Inggris sebagai bahasa internasional digunakan (Mydans, 2007).

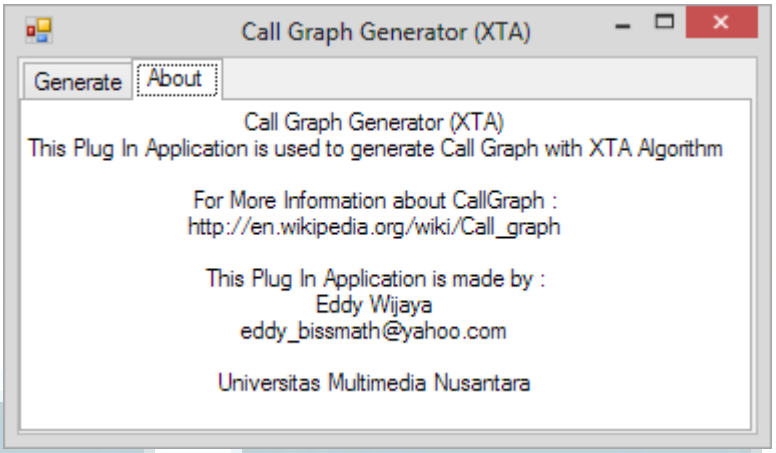
Aplikasi ini dibagi menjadi 2 bagian yang dipisahkan dengan tab. Tab pertama, yaitu "*Generate*" berisi komponen – komponen untuk menjalankan fungsi utama dari aplikasi (Generate Call Graph). Sedangkan tab yang kedua adalah "*About*", yang berisi tentang informasi seputar aplikasi ini.

Aplikasi dimulai dengan langsung secara otomatis mendeteksi project yang sedang dikerjakan oleh user dalam *workspace* eclipse. Pada tab "*Generate*" , *project-project* yang sudah didapat, akan langsung ditampilkan dalam sebuah *combo box*. Selanjutnya, setelah pengguna memilih *project* mana yang akan dibuat *call graph*-nya, pengguna bisa langsung menekan tombol "*Generate"* untuk menampilkan *Call Graph*.




Gambar 3.9 Desain antarmuka halaman *call graph*.

Gambar 3.9 adalah rancangan antarmuka halaman *call graph*. Halaman ini dibagi menjadi 2 bagian, bagian kiri merupakan tempat dimana *call graph* akan ditampilkan dan pada bagian ini pengguna dapat melakukan *drag and drop* pada *node-node* yang ada. Di bagian kanan, akan terdapat 2 buah tombol, yaitu "*Reset Position"* yang berfungsi untuk mengembalikan *node-node* pada posisi semula dan tombol "*Show Unused Method"* yang berfungsi untuk menampilkan daftar *method* yang ada pada *project* namun, tidak dipanggil oleh *method* apapun. Selain itu juga terdapat sebuah *label* yang berisikan tentang informasi dari *node* yang dipilih oleh pengguna. Informasi yang ditampilkan yaitu nama *method* yang dipilih, daftar *method* apa saja yang dipanggil olehnya dan *method* ini dipanggil oleh *method* apa saja.




Gambar 3.10 Desain Antarmuka Tab “About”.

Tab “About” berisi penjelasan singkat tentang apa aplikasi, tujuannya, pembuat dan hak cipta aplikasi. Tampilan desain antarmuka tab “About” dapat dilihat pada Gambar 3.10.

# BAB IV

# IMPLEMENTASI DAN UJI COBA

## 4.1 Implementasi Sistem

Sistem diimplementasikan berdasarkan rancangan yang telah dibuat, yaitu sebuah aplikasi *plug in* Eclipse. Implementasi dimulai dengan mendeklarasikan class *MainMenu* yang berfungsi untuk mendeteksi *Eclipse Project* yang ada dan membuat antarmuka untuk kemudian bisa tampil dan dipilih oleh pengguna.

Setelah tampilan antarmuka selesai dibuat, selanjutnya adalah membuat AST *(Abstract Syntax Tree*) yang bertujuan untuk mengambil dan menyimpan data-data dari *project*. Data-data tersebut antara lain pendeklarasian *class*, *method*, *field*, operasi *read/write* dan pemanggilan *method*.

Setelah memastikan data-data yang didapat sudah benar, dimulailah pembuatan *class* XTA dan beberapa *class* pendukung lainnya. *Class-class* ini berfungsi untuk memproses data-data yang ada menjadi *edges*, menggunakan Algoritma *Separate Type Analysis* (XTA).

Setelah melakukan uji coba untuk hasil *edges* yang dihasilkan, tahap terakhir adalah pengintegrasian fungsionalitas dari aplikasi *plug in* ini dengan antarmuka yang telah dibuat. *Edges* yang dihasilkan akan ditampilkan dalam bentuk *graph*, untuk selanjutnya dilakukan uji coba secara keseluruhan. Aplikasi *plug in* yang dibuat diberi nama “Call Graph Generator”.

### 4.1.1 Lingkup Implementasi

Lingkup implementasi menjelaskan piranti keras dan piranti lunak yang digunakan dalam tahap pengembangan aplikasi. Piranti keras yang digunakan untuk pemrograman adalah laptop ASUS seri N43SL dengan spesifikasi :

- Processor Intel® Core™ i7-2630QM CPU @ 2,00GHz
- Memori 4096 MB
- *Harddisk* 750GB

Spesifikasi piranti lunak yang digunakan pada laptop adalah sebagai berikut.

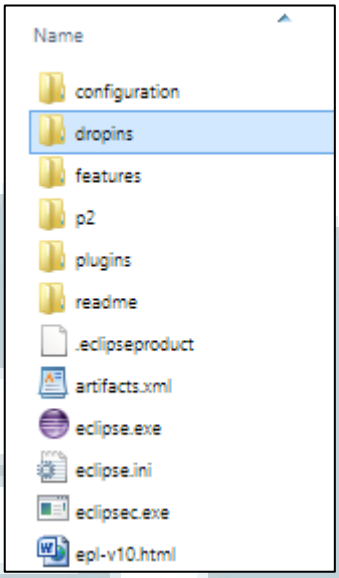
- Sistem operasi Windows 8 Enterprise
- Eclipse 4.3.0 Kepler
- Java Standard Edition dengan SDK versi 1.7
- Microsoft Office Word 2010
- Diagram Designer versi 1.26

### 4.1.2 Hasil Implementasi

Berikut ini adalah hasil implementasi sistem yang dirancang pada aplikasi *plug in Call Graph Generator (XTA).*

1. Instalasi

   Hasil dari aplikasi *plug-in* ini berupa *file* berekstensi (**.jar*). Untuk melakukan instalasi, pengguna hanya cukup meletakkan file *plug-in* pada *folder* “dropins” pada *folder* Eclipse.

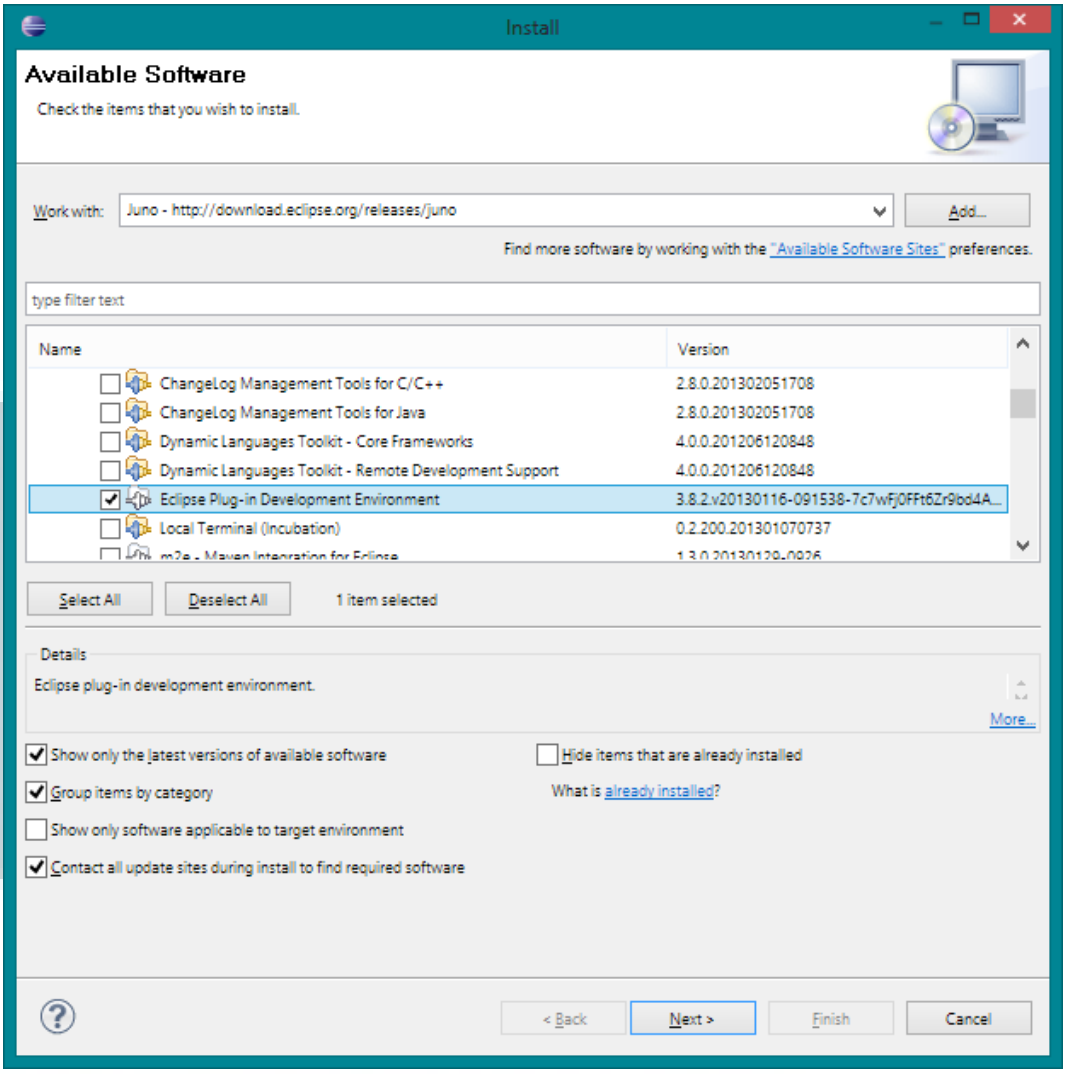



Gambar 4.1 Tampilan letak *folder* dropins.

Untuk bisa berjalan, aplikasi *plug-in* ini membutuhkan beberapa *plug-in* dan *package* yang harus sudah terinstalasi pada Eclipse pengguna, antara lain

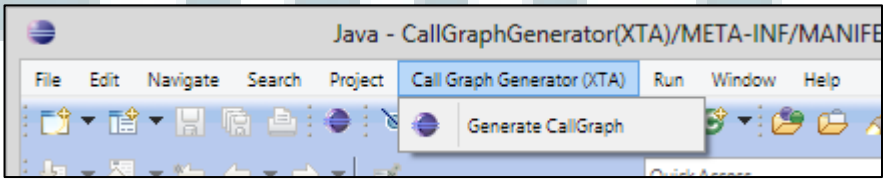
- org.eclipse.ui
- org.eclipse.core.runtime
- org.eclipse.jdt.core
- org.eclipse.core.resources

Jika, pada Eclipse pengguna tidak memiliki *plug-in* atau *package* yang disebutkan di atas, pengguna dapat melakukan instalasi dengan cara mengakses menu *Help > Install New Software*. Lalu, gunakan sumber dari url "*Juno - http://download.eclipse.org/releases/juno*" dan install *Eclipse Plug-in Development Environment* dalam *folder General Purpose Tools* seperti ditunjukkan pada Gambar 4.2 berikut ini.




Gambar 4.2 Tampilan menu instalasi piranti lunak Eclipse PDE

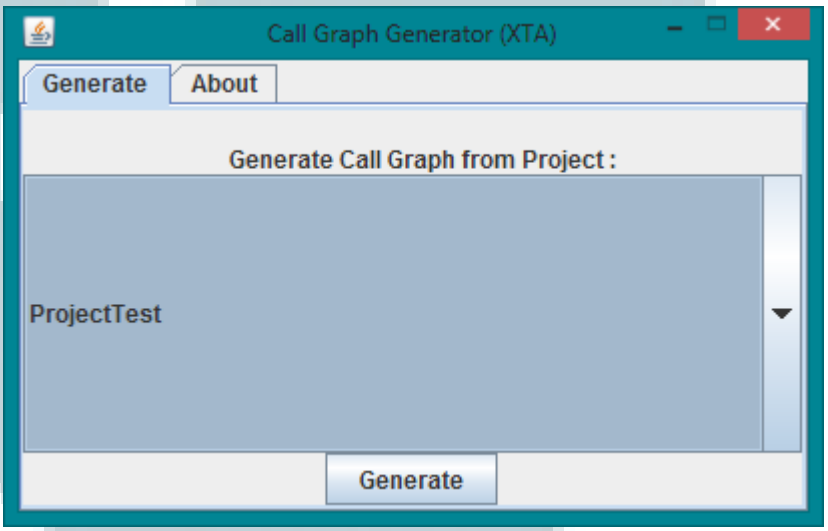
Jika persyaratan dari aplikasi *plug-in* sudah dipenuhi, setelah melakukan *restart* pada Eclipse, akan muncul sebuah menu *plug-in* bernama "*Call Graph Generator (XTA)*" pada Eclipse.




Gambar 4.3 Tampilan *plug in* pada *menu bar*.

2. Menu Utama

Jika menu *Generate Call Graph* dipilih, akan ditampilkan menu utama dari aplikasi *plug in*. Sesaat sebelum halaman ditampilkan, proses *getProjectsName* akan dijalankan dan hasilnya nama *project* yang ada pada *workspace* akan tampil pada *combobox*, untuk bisa dipilih.




Gambar 4.4 Tampilan menu utama aplikasi *plug in.*

3. *Extract data from Project*

Setelah memilih *project* dan menekan tombol *Generate*, proses *extract data from Project* akan dimulai. Proses ini akan membuat *Abstract Snytax Tree* (AST) dari *source code* yang ada pada *project*. Dengan pembuatan AST ini, informasi *class*, *method* dan *field* juga bisa didapatkan. Dalam proses pembuatan AST, setiap *method* dalam *source code* akan dikunjungi dan pada saat itulah data tersebut bisa diambil, dengan cara memodifikasi *class ASTVisitor*, yaitu dengan

membuat sebuah *class* yang melakukan *inheritance* terhadap class *ASTVisitor* dan melakukan *override* terhadap *method visit*, data yang dibutuhkan bisa diambil.

```
public class MethodVisitor extends ASTVisitor {
    Map<MethodDeclaration, MethodBundle> map = new HashMap<MethodDeclaration, MethodBundle>();
    MethodBundle temp;
    ClassHierarchy ch = new ClassHierarchy();

    private boolean inVariableAssignment;

    public boolean visit(MethodDeclaration node)
    {
        boolean isStatic = true;
        String kelas = node.resolveBinding().getDeclaringClass().getName();
        String parent = node.resolveBinding().getDeclaringClass().getSuperclass().getName();
        if(Modifier.isStatic(node.resolveBinding().getModifiers()))
          isStatic = true;

        Kelas k = ch.new Kelas(kelas, parent, isStatic);
        ClassHierarchy.addClass(k);

        temp = new MethodBundle();
        return super.visit(node);
    }

    public void endVisit(MethodDeclaration node)
    {
          map.put(node, temp);
          temp = null;
    }
```

Gambar 4.5 Potongan s*ource code class MethodVisitor*.

Setelah AST sudah berhasil dibuat, data-data tersbut akan dikumpulkan dalam sebuah *HashMap* dan disimpan sebagai modal untuk proses berikutnya.

Jika *source code* yang diproses adalah seperti yang ditunjukkan pada Gambar 4.6, maka setelah melalui proses *extract data*, data yang didapat adalah seperti yang ditunjukkan pada Gambar 4.7.

```
public class Main {

    public static void main(String[] args)
    {
        B b1 = new B();
        A a2 = new A();

        f(b1);
        g(b1);
    }

    static void f(A a2)
    {
        a2.foo();
    }

    static void g(B b3)
    {
        B b4 = b3;
        b4 = new C();
        b4.foo();
    }

}
```

```
class A
{
    void foo(){}    //A.foo
}

class B extends A
{
    void foo(){}    //B.foo
}

class C extends B
{
    void foo(){}    //C.foo
}

class D extends B
{
    void foo(){}    //D.foo
}
```

Gambar 4.6 *Source code* dari *project* "*ProjectTest*".

```
C.foo
Static = false
    Memanggil :

B.foo
Static = false
    Memanggil :

Main.main
Static = true
    Memanggil :
        - Main.f
        - Main.g

D.foo
Static = false
    Memanggil :
```

```
Main.f
Static = true
    Memanggil :
        - A.foo
A.foo
Static = false
    Memanggil :

Main.g
Static = true
    Memanggil :
        - B.foo
```

Gambar 4.7 Data *method* yang diambil dari *project* "*ProjectTest*".

4. *Generate Call Graph*

Setelah data didapat, proses selanjutnya adalah proses *generate call graph* menggunakan algoritma RA dan XTA. Algoritma RA juga sering disebut sebagai *name based resolution*, karena hanya mengandalkan nama dalam menentukan *method* mana yang memanggil *method* mana.

```
public void generateCallGraph(Method entryPoint)
{
    MethodUtility mu = new MethodUtility();
    reachable.add(entryPoint);

    int i = 0;

    for(i = 0; i<reachable.size(); i++)
    {
        Method thisMethod = reachable.get(i);
        for(Method m : MethodUtility.methodList)
        {
            //kalo method dlm reachable ada panggil method lain...
            if(MethodUtility.isMethodCallingbyName(thisMethod, m))
            {
                //masukin ke reachable..
                reachable.add(m);
                //buat edge...
                mu.addNewEdge(thisMethod.getText(), m.getText());
            }

        }
    }
}
```

Gambar 4.8 *Source code method generateCallGraph* (RA).

Untuk setiap *method*, dimulai dari *method main*, dilakukan pengamatan untuk mencari tahu apakah ada aktivitas pemanggilan *method*. Jika ada, maka akan dicari *method* dengan nama tersebut dalam daftar dan dibuat *edge*-nya.

Contohnya berdasarkan data yang didapat dari hasil ekstrak data *source code* "*ProjectTest*", jika dibuat *call graphnya* menggunakan algoritma RA, prosesnya adalah sebagai berikut

1. Proses dimulai dari *entry point*, yaitu Main.main yang memanggil *method* f dan g. Lalu, dari keseluruhan data yang didapat, akan dicari method yang bernama "f" dan "g".
2. Ditemukanlah Main.f dan Main.g. Kedua *method* tersebut akan masuk kedalam *array* reachable, dan terbentuk *edge* [Main.main-> Main.f] dan [Main.main-> Main.g].
3. Selanjutnya proses akan berganti ke *method* selanjutnya dalam *array* reachable, yaitu Main.f. *Method* ini memanggil *method* foo, maka akan dicari *method* yang bernama "foo".
4. Sehingga didapatlah empat *method*, yaitu A.foo, B.foo, C.foo dan D.foo. Ke-empat *method* ini akan dimasukkan kedalam *array* reachable, dan dibuat edgenya dari Main.f menuju ke-empat *method* tersebut.

Proses tersebut akan berlanjut hingga dihasilkan 10 buah edge sebagai berikut

- Main.main -> Main.f dan Main g
- Main.f -> A.foo, B.foo, C.foo dan D.foo
- Main.g -> A.foo, B.foo, C.foo dan D.foo

Algoritma XTA secara teori memerlukan memori yang besar dan proses yang lebih rumit namun, bisa jadi prosesnya bisa lebih cepat. Sama seperti RA, XTA dimulai dengan melakukan proses terhadap *method* main. Pengamatan yang dilakukan pada setiap *method* adalah sebagai berikut

- Aktivitas pendeklarasian *new instance*

```
//=== jika new ================================================
for(NewInstance ni : m.listInstance)
{
    m.add_sList(ni.tipe);
    //cari tau yg di new itu, ada constructornya nga?
    Method methodNew = MethodUtility.getMethodWithName(ni.tipe, ni.tipe);
    if(methodNew != null)
        addReachable(methodNew);
}
//=============================================================
```

Gambar 4.9 *Source code* pengecekan aktivitas *new* pada *method* (XTA).

Jika ditemukan ada aktivitas *new*, maka tipe dari variabel *new* akan dimasukkan kedalam *sList* dari *m*. Setelah itu jika *class* yang di *new* memiliki *constructor*, maka *contructor* akan masuk kedalam *array reachable*.

- Aktivitas *read* dan *write field*

Jika ditemukan ada aktivitas *read*, tipe data dari *field* yang di-*read* akan dimasukkan kedalam *sList* milik *m*. Sebaliknya, jika ditemukan ada aktivitas menulis *field* akan dicari irisan antara *subclass* dari *field* yang ditulis dan *sList* dari m untuk kemudian dimasukkan kedalam *sList* dari *field*.

Untuk lebih jelas, Gambar 4.10 merupakan sebuah bagian *source code* dari algoritma XTA yang bertugas dalam pemrosesan aktivitas *read* dan *write* dalam sebuah *method*.

```
//4. field yg dibaca
for(String field : m.listRead)
{
    Field f = MethodUtility.getField(mu.new Field(field));
    if(f != null)
    {
        m.add_sList(field);
        System.out.println("\t\t (read)Masukkin kedalem sList : " + field);
    }
}

//5. field yg di tulis
for(String field : m.listWrite)
{
    Field f = MethodUtility.getField(mu.new Field(field));
    for(String subType : ClassHierarchy.getSubClass(field))
    {
        for(String sm : m.sList)
        {
            if(sm.equals(subType))
            {
                f.add_sList(field);
            }
        }
    }
}
```

Gambar 4.10 *Source code* pengecekan aktivitas *read* dan *write field* (XTA)

Fungsi *getSubClass* yang ada pada Gambar 4.10 dijelaskan prosesnya pada Gambar 4.11. Fungsi ini sedikit berbeda dari terminologi *subclass* yang berarti kelas-kelas turunan dari kelas tersebut. Fungsi *getSubClass* yang digunakan dalam algoritma XTA ini tidak akan hanya mengambil kelas anak saja, tetapi juga semua *superclass*-nya, hingga kelas sebelum kelas *Object*.

```
public static ArrayList<String> getSubClass(String target)
{
    ArrayList<String> temp = new ArrayList<String>();
    Kelas node = getKelas(target);
    if(node == null)
        return temp;

    //naik sampai ke anak Object
    while((node != null) && (!node.parent.equals(ROOT)))
    {
        node = getKelas(node.parent);
    }

    if(temp.indexOf(node.anak) == -1)
        temp.add(node.nama);

    //turun sampe ujung
    visitAnak(temp, node);
    return temp;
}
```

Gambar 4.11 *Source code* untuk mendapatkan *sub-class*.

- Aktivitas pemanggilan *method*

Jika ditemukan bahwa terdapat panggilan *method*, *subclass* tipe dari kelas *method* yang dipanggil akan dibandingkan dengan *sList* dari m, sehingga bisa didapatkan *method-method* yang mungkin dipanggil, untuk kemudian bisa dibuat *edgenya* [*m->kelas.methodDipanggil*] dan juga *methodDipanggil* akan dimasukkan kedalam *array Reachable*. Untuk lebih jelasnya bisa melihat potongan *source code* yang ditampilkan pada Gambar 4.12.

```
for(PanggillanMethod pm : m.listPanggilan)
{
    System.out.println("\t ================================");
    System.out.println("\t method yang dipanggil :" + pm.kelas + "." + pm.nama);
    System.out.println("\t\t Subclass dari class ini : ");
    for(String sub : ClassHierarchy.getSubClass(pm.kelas))
    {
        System.out.println("\t\t subClass : " + sub);
    }

    System.out.println("\t\t S dari method ini : ");
    System.out.println("\t\t ============================");

    for(String s : m.sList)
    {
        System.out.println("\t\t S : " + s);
        Method methodDipanggil = null;
        methodDipanggil = MethodUtility.getMethodWithName(s, pm.nama);

        if(methodDipanggil == null)
        {
            System.out.println("\t\t Method " + s + "." + pm.nama + " = null");
            continue;
        }
        System.out.println("\t\t Method " + s + "." + pm.nama + " = ada");


        //=== cek SList ==========================================
        for(String sub : ClassHierarchy.getSubClass(s))
        {
            System.out.println("\t\t subClassnya " + s + " : " + sub);
            if(s.equals(sub))
            {
                //add new EDGE
```

Gambar 4.12 Potongan *source code* pemrosesan pemanggilan *method* (XTA) bagian 1.

Setelah itu, untuk membuat daftar *sList* menjadi lebih akurat, tipe *return* dari *method* yang dipanggil dan juga *parameter* yang dikirim harus juga diproses. Untuk pengiriman *parameter*, *subtype* dari *parameter* akan di iriskan dengan *sList* milik *m* dan hasilnya akan dimasukkan kedalam *sList* milik *methodDipanggil*. Untuk *return* dari methodDipanggil, *subtype* dari *return* tersebut akan dicari irisannya dengan *sList* dari *methodDipanggil* dan hasilnya akan dimasukkan kedalam *sList* dari *m*. Berikut adalah potongan *source code* yang

menunjukkan pemrosesan *parameter* dan *return* dari panggilan *method*.

```
//=== cari irisan parameter ==============================
for(String param : methodDipanggil.parameters)
{
    System.out.println("\t\t parameter M' : " + param);
    for(String subType : ClassHierarchy.getSubClass(param))
    {
        for(String sm : m.sList)
        {
            if(subType.equals(sm))
            {
                methodDipanggil.add_sList(sm);
                System.out.println("\t\t (param)Masukkin kedalem sList : " + sm);
            }
        }
    }
}
//=== cari irisan return ================================
for(String sm : methodDipanggil.sList)
{
    for(String subType : ClassHierarchy.getSubClass(methodDipanggil.returnType))
    {
        if(subType.equals(sm))
        {
            m.add_sList(sm);
            System.out.println("\t\t (return)Masukkin kedalem sList : " + sm);
        }
    }
}
```

Gambar 4.13 Potongan *source code* pemrosesan pemanggilan *method* (XTA) bagian 2.

Contohnya berdasarkan data yang didapat dari hasil ekstrak data *source code* "*ProjectTest*", jika dibuat *call graphnya* menggunakan algoritma XTA, prosesnya adalah sebagai berikut

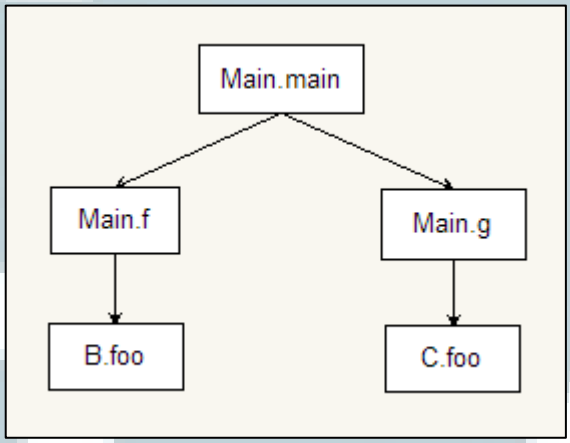
1. Proses dimulai dari *entry point*, yaitu Main.main.
2. Main.main melakukan aktivitas pendeklarasian *new instance*, sehingga *class* akan di masukkan kedalam sList. Namun, karena

tidak ditemukan *constructor* pada *class* yang di *new*, maka aktivitas ini tidak diproses lebih lanjut.

3. Main melakukan aktivitas menulis *field* B, sehingga irisan dari *subclass* B (yaitu A, B, C dan D) dengan sList dari m (yaitu A dan B) dimasukkan kedalam sList dari *field* (yaitu A dan B).
4. Setelah memasukkan semua *method* yang *static* kedalam sList, pemrosesan pemanggilan *method* dimulai. Main.main terdata telah melakukan panggilan terhadap *method* Main.f. Lalu, *subclass* dari Main, dibandingkan dengan SList m (yaitu A, B, dan Main) jika sama (misalnya *class* yang sama disebut *class* X), maka akan dicari keberadaan *method* "X.f". Jika ada, *method* tersebut akan dimasukkan kedalam *array* reachable, dan dibuat *edge*-nya (dalam kasus ini, *class* yang sama (X) adalah Main) [Main.main -> Main.f]
5. Proses berlanjut dengan mencari irisan *subtype parameter* dari Main.f (yaitu A, B, C, D) dengan SList m (yaitu A, B, Main) untuk dimasukkan kedalam sList milik *method* Main.f. Karena *method* Main.f tidak memiliki *return type* (*void*), pemrosesan aktivitas pemanggilan *method* selesai, dilanjutkan dengan memproses pemanggilan Main.g
6. Proses tersebut akan berlanjut hingga dihasilkan 7 buah *edge* yaitu
   - Main.main -> Main.f dan Main.g
   - Main.f -> A.foo dan B.foo

- Main.g -> A.foo, B.foo dan C.foo

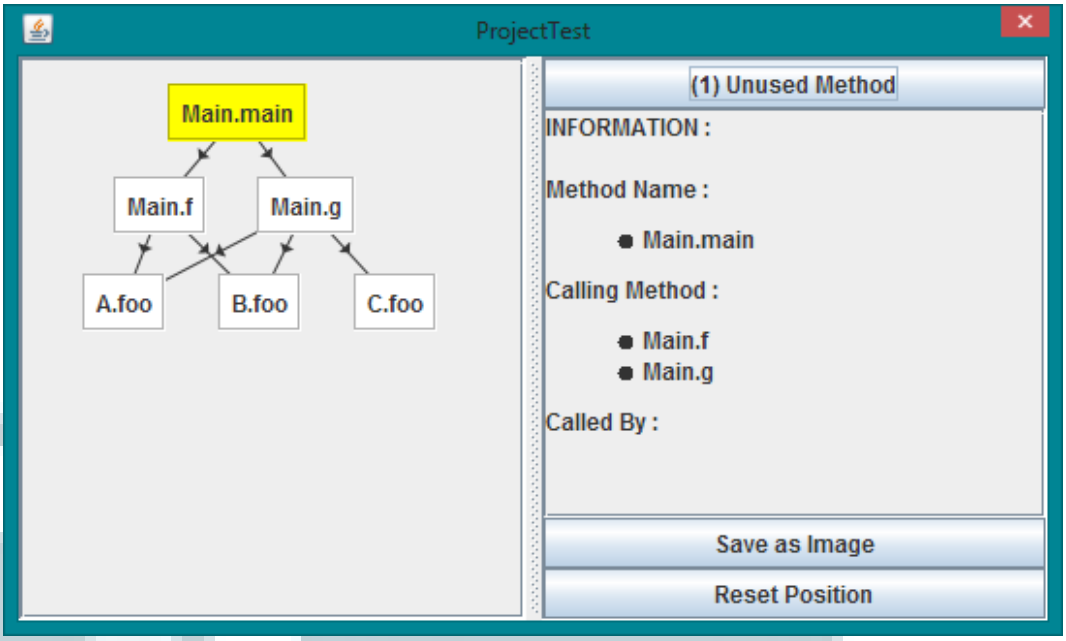
Jika dibandingkan dengan hasil analisa *source code* secara manual, *edge* yang dihasilkan berjumlah 5 (lima buah), lebih sedikit dibandingkan dengan XTA yang berjumlah 7 dan RA yang berjumlah 10. Call graph yang didapat dengan analisa source code "ProjectTest" secara manual adalah sebagai berikut



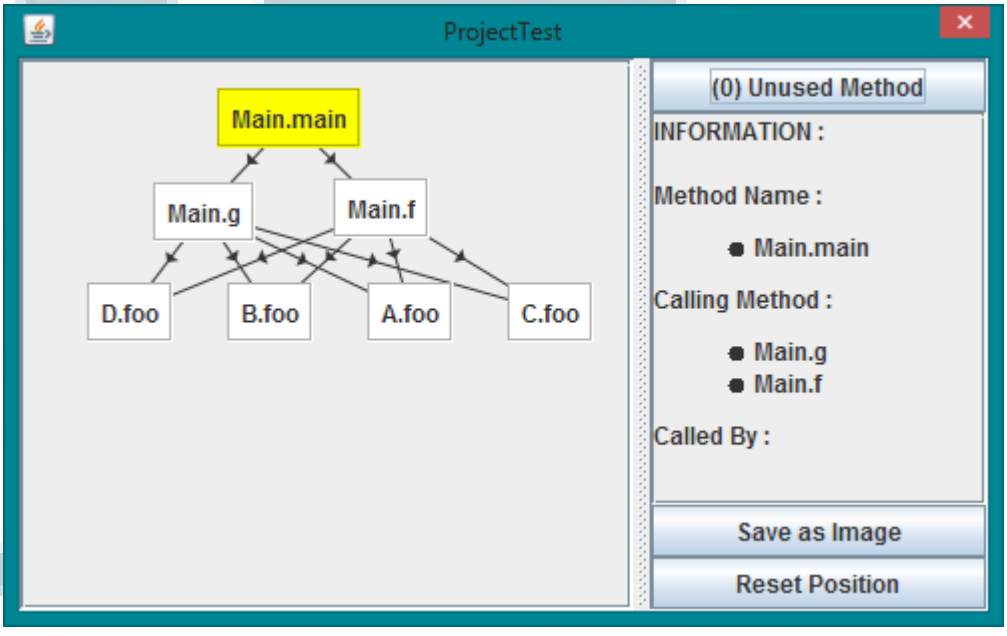

Gambar 4.14 Call graph analisa manual dari source code "ProjectTest".

5. *DrawGraph*

Setelah *edge-edge* didapat, langkah selanjutnya adalah memvisualisasikan data tersebut menjadi sebuah grafik. Proses ini dinamakan sebagai *DrawGraph*. *Edge–edge* tersebut pertama akan disusun berdasarkan *level* kedalaman dihitung dari *entry point*, lalu setelah ditentukan posisi dasarnya, barulah digambar ke *panel*.

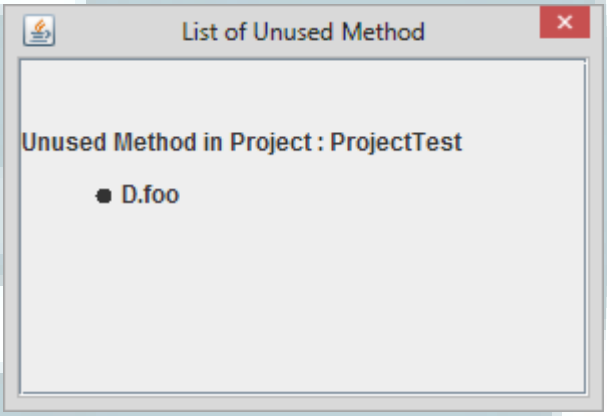



Gambar 4.15 Tampilan hasil *call graph* yang di generate menggunakan algoritma XTA.




Gambar 4.16 Tampilan hasil *call graph* yang di generate menggunakan algoritma RA

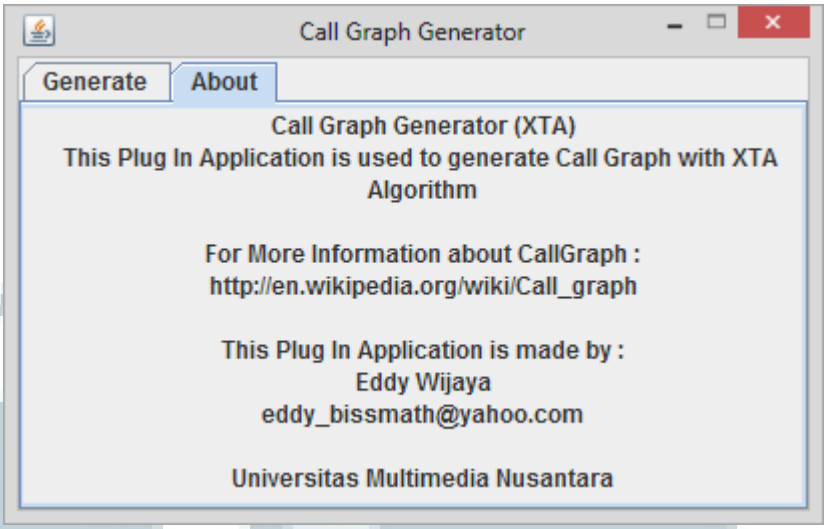
Untuk melihat informasi yang lebih lengkap, *node* bisa dipilih. Selain itu, *node* juga bisa dipindahkan posisinya dengan *drag and drop* sesuai dengan keinginan pengguna. Di bagian kanan terdapat tombol *reset position*, untuk mengembalikan *node* pada posisi awal dan tombol *Unused Method* untuk melihat daftar *method* yang tidak terpakai pada *project*.




Gambar 4.17 Tampilan daftar *method* yang tidak terpakai.

6. *About*

Halaman *about* berada pada *tab* kedua pada halaman menu utama. Halaman *about* berisi tentang deskripsi singkat dari aplikasi plug in.




Gambar 4.18 Tampilan halaman *About.*

## 4.2 Uji Coba

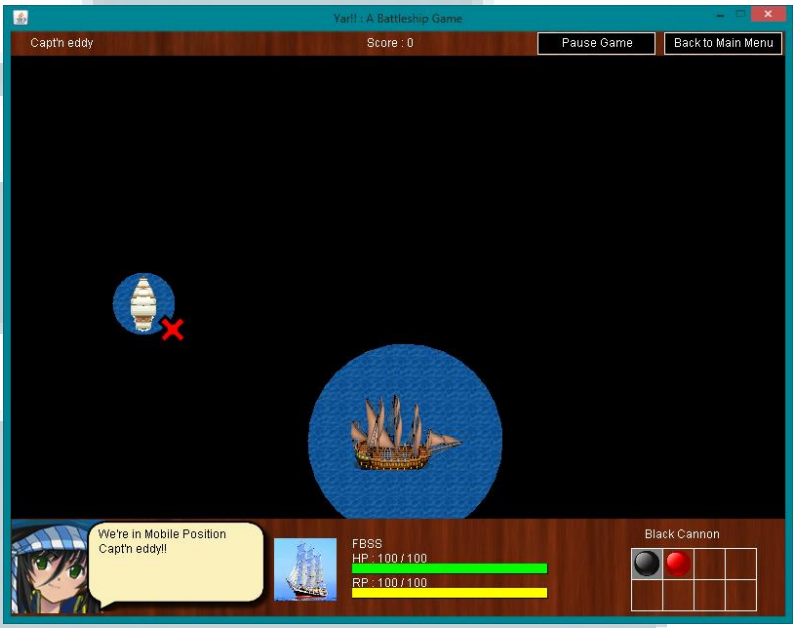
Uji coba dilakukan untuk mengukur seberapa baik fungsionalitas aplikasi plug in Call Graph generator dengan menggunakan algoritma XTA. Uji coba yang dilakukan adalah pengujian *user experience* menggunakan kuesioner, uji performa dan pengujian validitas dari keluaran yang dihasilkan dengan *sampling*. Pengujian *user experience* dilakukan dengan menyebar kuesioner kepada beberapa orang yang memiliki pengetahuan tentang pemrograman. Sedangkan untuk uji performa, akan dilakukan berdasarkan 4 variabel, yaitu memori yang digunakan, lama waktu proses, jumlah *node* dan *edge* yang dihasilkan. Pengujian validitas keluaran dilakukan dengan mengambil beberapa bagian dari sebuah *source code* sebagai sampel, untuk dianalisis secara manual kemudian dibandingkan hasilnya dengan keluaran aplikasi.

### 4.2.1 Data Uji Coba

Uji coba dilakukan dengan 3 jenis *source code*, yaitu

1. *Source code* berukuran kurang lebih 4000 *line of code*.

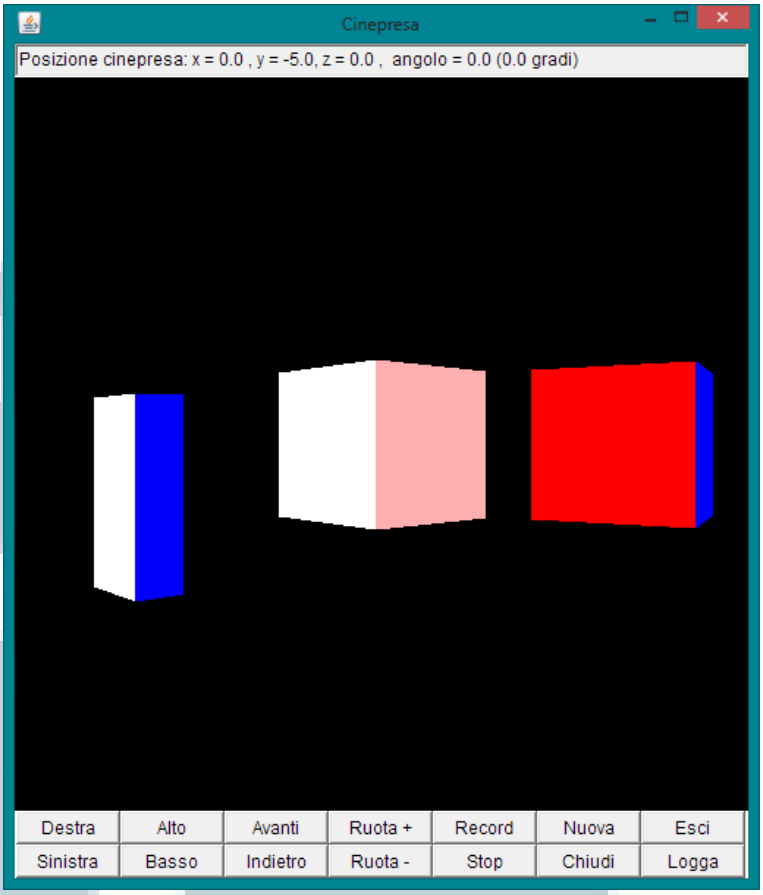
   *Source code* yang digunakan berasal dari sebuah permainan yang menggunakan bahasa JAVA, yaitu "*Yar!! : A Battleship Game*".




Gambar 4.19 Tampilan *screenshot* game "Yar!! : A Battleship Game".

2. *Source code berukuran* kurang lebih 500 *line of code*

   *Source code* berasal dari sebuah *project* JAVA yang dibuat oleh Giacomo Mazzocato bernama "*Cinepresa*" yang bertujuan untuk menujukkan bentuk geometri secara 3D.




Gambar 4.20 Tampilan *screenshot* aplikasi "*Cinepresa*".

3. Source code sederhana

   Adalah sebuah *source code* sederhana yang menggunakan banyak sifat *inheritance* yang secara teori dapat membedakan *output* antara algoritma RA dengan XTA. *Source code* ini dinamakan "*TestCode*".

```
public class Main {

    public static void main(String[] args)
    {
        B b1 = new B();
        A a2 = new A();

        f(b1);
        g(b1);
    }

    static void f(A a2)
    {
        a2.foo();
    }

    static void g(B b3)
    {
        B b4 = b3;
        b4 = new C();
        b4.foo();
    }

}
```

```
class A
{
    void foo(){}    //A.foo
}

class B extends A
{
    void foo(){}    //B.foo
}

class C extends B
{
    void foo(){}    //C.foo
}

class D extends B
{
    void foo(){}    //D.foo
}
```

Gambar 4.21 *Source code* "*TestCode*".

### 4.2.2 Skenario Uji Coba

Uji coba yang dilakukan mengikuti skenario yang masing-masing digunakan untuk menilai *parameter-parameter* yang ada. Skenario tersebut adalah sebagai berikut.

1. Uji coba performa *generate call graph* dengan ketiga *source code* dan algoritma RA dan XTA. *Parameter* yang ingin dinilai adalah bagaimana skalabilitas dari kedua algoritma, dinilai dari waktu dan memori yang dibutuhkan. Selain itu juga dinilai tingkat akurasi dalam menghasilkan *call graph* dilihat dari jumlah *node* dan *edge* yang dihasilkan.
2. Uji coba validitas keluaran akan dilakukan dengan mengambil 5 buah *method* dari *Yar : A Battleship Game* sebagai sampel, untuk kemudian dianalisis secara manual dan dibandingkan dengan hasil keluaran program.

3. Uji coba *user experience* akan dilakukan dengan membagi kuesioner dengan pertanyaan berskala 1-5 (*likert scale*), mengenai respon terhadap kemudahan penggunaan dan hasil *output* dari aplikasi CallGraph Generator. Pertanyaan dalam kuesioner dibuat dengan mereferensi kepada contoh kuesioner *Software Usability Measurement Inventory* (SUMI) dan akan dibagikan kepada dua puluh responden yang memiliki pengetahuan tentang pemrograman.

### 4.2.3 Hasil dan Evaluasi Uji Coba

Hasil uji coba dijelaskan satu per satu berdasarkan poin-poin skenario yang sudah ditentukan sebagai berikut.

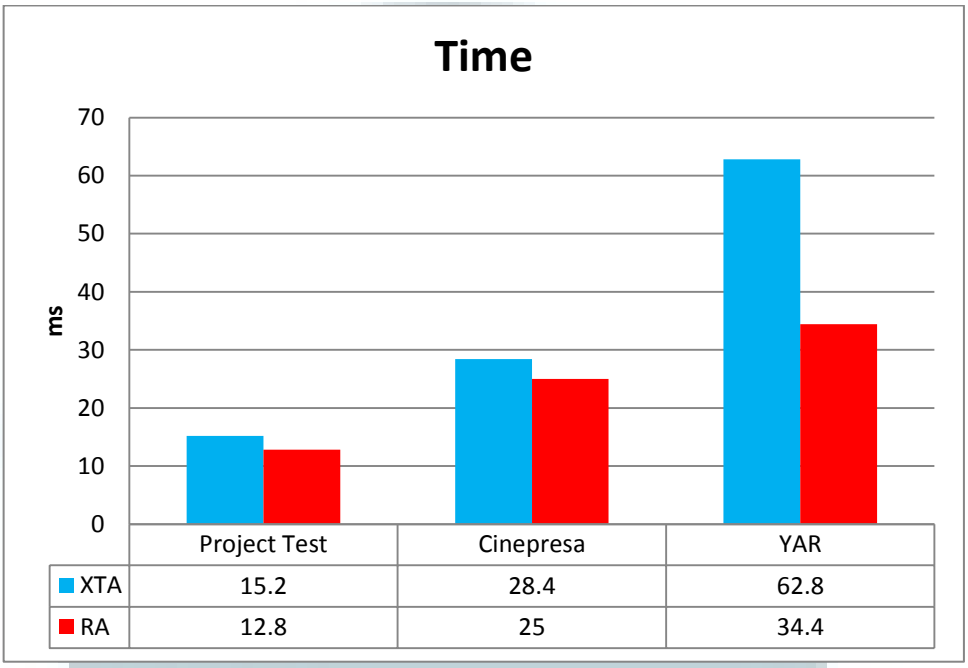
1. Uji performa

   Tabel berikut merupakan hasil uji coba waktu yang dibutuhkan untuk melakukan *generate call graph* dengan *ketiga source code* yang ada menggunakan algoritma XTA dan RA.

Tabel 4.1 Waktu yang dibutuhkan untuk menghasilkan *call graph.*

| *Source Code* | Algoritma | Waktu Pemrosesan (*milisecond*) | | | | | Rata-rata (*milisecond*) |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | |
| *Yar : A Battleship Game* | XTA | 70 | 63 | 63 | 56 | 62 | 62.8 |
| | RA | 36 | 33 | 34 | 34 | 35 | 34 |
| *Cinepresa* | XTA | 29 | 31 | 28 | 27 | 27 | 28.4 |
| | RA | 25 | 25 | 24 | 24 | 27 | 25 |
| *TestCode* | XTA | 15 | 14 | 15 | 15 | 17 | 15.2 |
| | RA | 12 | 13 | 13 | 13 | 13 | 12.8 |

Hubungan antara besarnya *source code* dan waktu pemrosesan *call graph* ditunjukkan dalam diagram berikut ini.




Gambar 4.22 Diagram hubungan antara waktu yang dibutuhkan dengan besar *source code*.

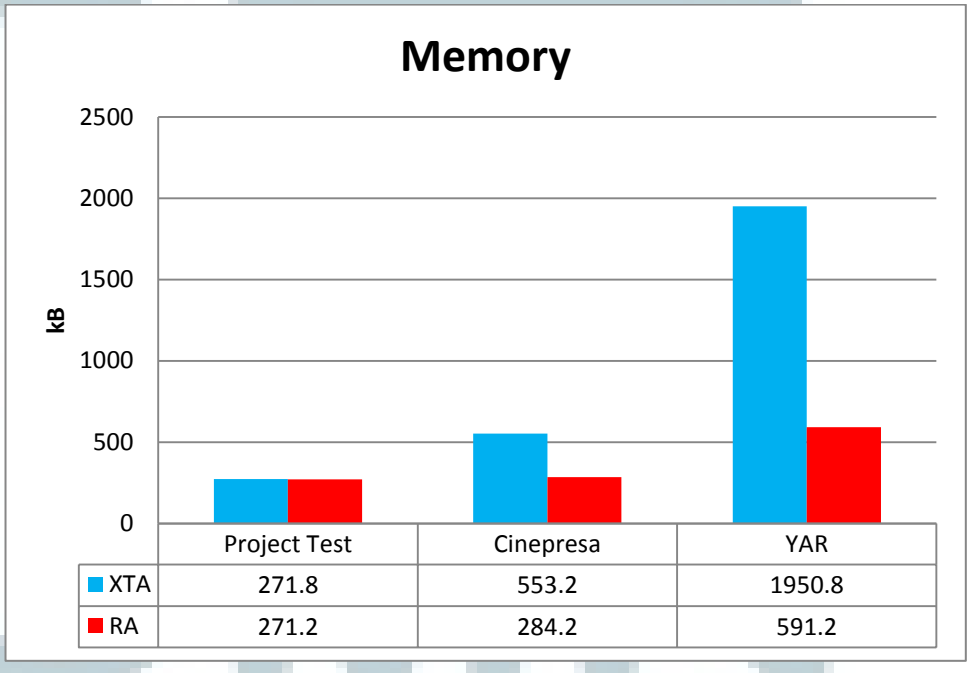
Dari diagram diatas, dapat disimpulkan bahwa semakin besar *source code*, waktu yang diperlukan untuk pemrosesan *generate call graph* semakin besar pula. Khususnya pada algoritma XTA, peningkatan waktu yang dibutuhkan dalam memproses *project Cinepresa* ke *YAR* adalah sebanyak 121%. Bisa dibilang cukup drastis jika dibandingkan peningkatan waktu yang dialami algoritma RA yang hanya sebanyak 37.6%.

Tabel berikut merupakan hasil uji coba waktu yang dibutuhkan untuk melakukan *generate call graph* dengan *ketiga source code* yang ada menggunakan algoritma XTA dan RA.

Tabel 4.2 Memori yang dibutuhkan untuk menghasilkan *callgraph.*

| Source Code | Algoritma | Memori yang dibutuhkan (KB) | | | | | Rata-rata (KB) |
|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | |
| Yar : A Battleship Game | XTA | 1990 | 1933 | 1993 | 1963 | 1965 | 1950.8 |
| | RA | 599 | 598 | 588 | 586 | 585 | 591.2 |
| Cinepresa | XTA | 278 | 268 | 276 | 280 | 319 | 553.2 |
| | RA | 576 | 535 | 554 | 580 | 521 | 284.2 |
| TestCode | XTA | 266 | 266 | 267 | 304 | 256 | 271.8 |
| | RA | 273 | 267 | 249 | 264 | 303 | 271.2 |

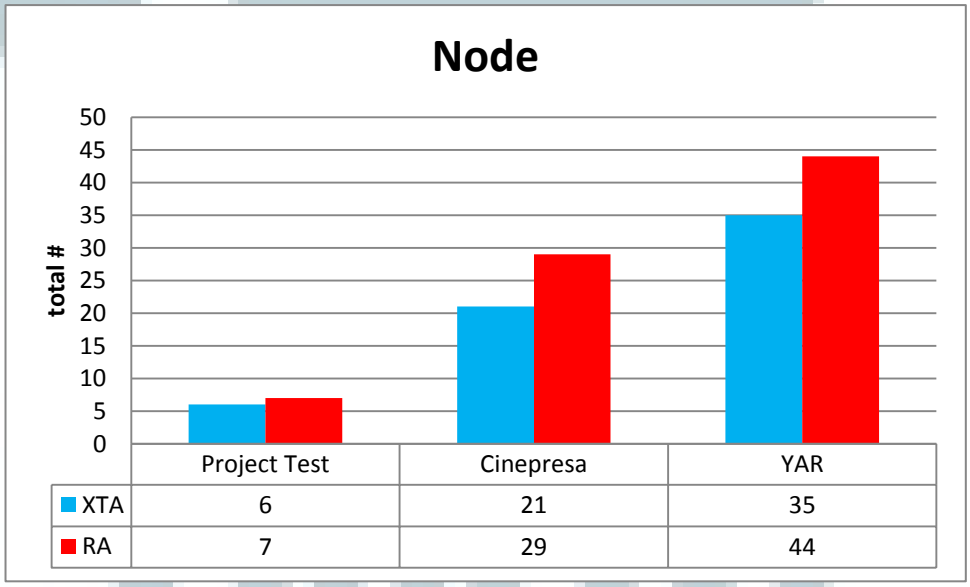
Hubungan antara besarnya *source code* dan waktu pemrosesan *call graph* ditunjukkan dalam diagram berikut ini.



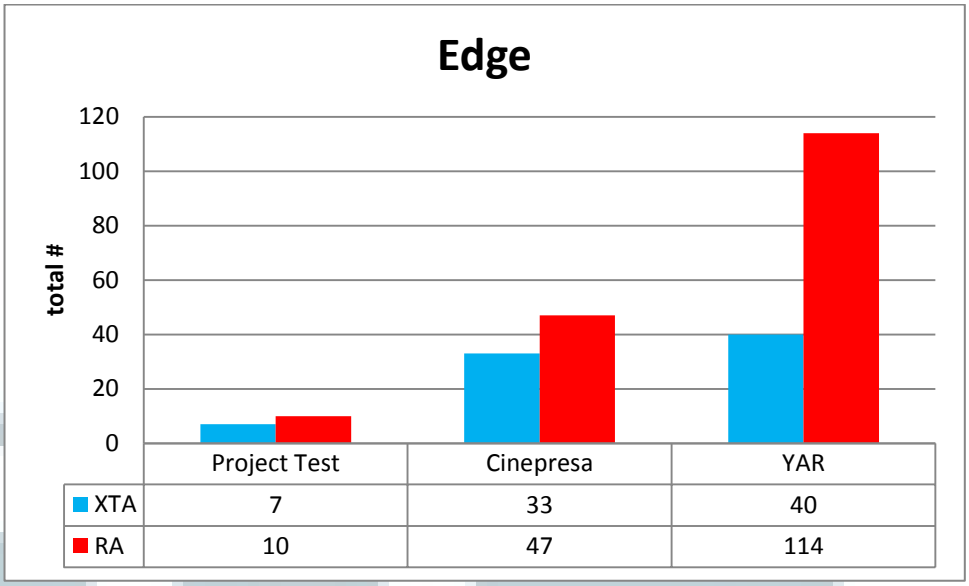

Gambar 4.23 Diagram hubungan antara memori yang dibutuhkan dengan besar *source code.*

Dari diagram tersebut, dapat disimpulkan bahwa semakin besar *source code*, memori yang diperlukan untuk pemrosesan *generate call graph* semakin besar pula. Sama halnya seperti waktu yang diperlukan, peningkatan yang cukup drastis dialami oleh algoritma XTA pada *project Cinepresa* dan *YAR* yaitu sebanyak 252% dibandingkan dengan RA yang 108%. Hal ini disebabkan karena algoritma XTA membutuhkan sebuah *list* pada setiap *method* dan *field*-nya.

Grafik berikut menjelaskan hasil uji coba yang dibutuhkan untuk melakukan *generate call graph* dengan ketiga *source code* yang ada menggunakan algoritma XTA dan RA.




Gambar 4.24 Diagram hubungan antara *node* yang dihasilkan dengan besar *source code*.




Gambar 4.25 Diagram hubungan antara *edge* yang dihasilkan dengan besar *source code*.

Berdasarkan kedua grafik diatas, dapat disimpulkan bahwa XTA memiliki akurasi yang lebih tinggi (dilihat dari *node* dan *edge* yang lebih sedikit). Dengan lebih sedikitnya jumlah *node* dan *edge*, *call graph* yang dihasilkan akan tampak lebih sederhana.

2. Uji validitas keluaran

Tabel berikut menunjukkan hasil analisis secara manual dan keluaran dari implementasi algoritma XTA dan RA dalam aplikasi Call Graph Generator. Data dalam tabel merupakan method *sampling* dari *source code YAR*, yang digunakan untuk pengujian validitas aplikasi Call Graph Generator.

Tabel 4.3 Daftar edge yang dihasilkan oleh beberapa method sampel.

| Manual | XTA | RA |
|---|---|---|
| **Method : MainGamePanel** | | |
| • Craft.Craft<br>• Board.Board<br>• GameMenu.GameMenu<br>• MainGamePanel.generate Enemy<br>• TAdapter.TAdapter | • Craft.Craft<br>• Board.Board<br>• GameMenu.GameMenu<br>• MainGamePanel.generate Enemy<br>• TAdapter.TAdapter<br>• Craft.getImage | • Craft.Craft<br>• Board.Board<br>• GameMenu.GameMenu<br>• MainGamePanel.generate Enemy<br>• TAdapter.TAdapter<br>• Craft.getImage<br>• EffectAnimation.getImage<br>• Missile.getImage<br>• enemy.getImage<br>• Tracker.getImage |
| **Method : Board.Board** | | |
| • BoardMouseListener. BoardMouseListener | • BoardMouseListener. BoardMouseListener | • Craft.getImage<br>• EffectAnimation.getImage<br>• Missile.getImage<br>• enemy.getImage<br>• Tracker.getImage |
| **Method : ReplayPlayer.ReplayPlayer** | | |
| • dal_Replay.dal_Replay<br>• BoardMouseListener. BoardMouseListener<br>• ReplayPlayer.getImage Scaled<br>• MainGamePanel.Main GamePanel<br>• MainGamePanel.getStage<br>• MainGamePanel.get PlayerName<br>• MainGamePanel.setBoard Switch | • dal_Replay.dal_Replay<br>• BoardMouseListener. BoardMouseListener<br>• dal_Replay.getImageScaled<br>• MainGamePanel.Main GamePanel<br>• MainGamePanel.getStage<br>• MainGamePanel.get PlayerName<br>• MainGamePanel.setBoard Switch | • dal_Replay.dal_Replay<br>• BoardMouseListener. BoardMouseListener<br>• dal_Replay.getImage Scaled<br>• MainGamePanel.Main GamePanel<br>• MainGamePanel.getStage<br>• MainGamePanel.get PlayerName<br>• MainGamePanel.setBoard Switch<br>• Craft.getImage<br>• EffectAnimation.getImage<br>• Missile.getImage<br>• enemy.getImage<br>• Tracker.getImage |

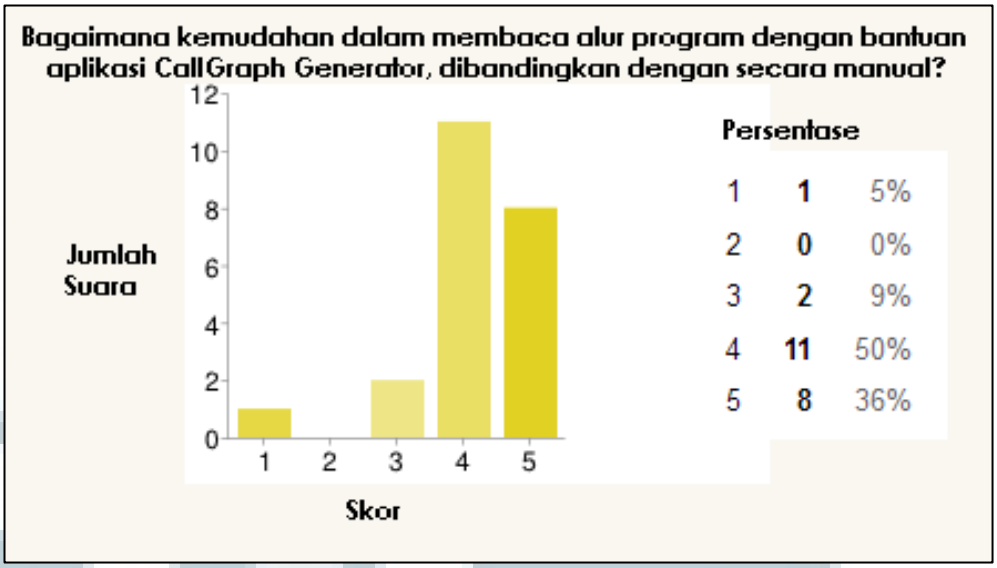
Tabel 4.4 Daftar edge yang dihasilkan oleh beberapa method sampel (lanjutan).

| **Method : Form_InputPlayerName** | | |
|---|---|---|
| • InputAdapter.Input Adapter | • InputAdapter.Input Adapter | • InputAdapter.Input Adapter<br>• dal_Replay.dal_Replay<br>• Craft.getImage<br>• EffectAnimation.getImage<br>• Missile.getImage<br>• enemy.getImage<br>• Tracker.getImage |
| **Method : enemy.enemy** | | |
| • enemy.setDir | • enemy.setDir<br>• enemy.getImage | • enemy.setDir<br>• Craft.setDir<br>• Craft.getImage<br>• EffectAnimation.getImage<br>• Missile.getImage<br>• enemy.getImage<br>• Tracker.getImage |

Berdasarkan tabel diatas, keluaran hasil dari algoritma XTA dan RA yang diimplementasikan ke dalam Call Graph Generator adalah *valid*. Hal tersebut ditunjukkan dengan semua hasil dari analisis manual yang tampak muncul dalam keluaran dari algoritma XTA dan RA yang diimplementasikan ke dalam Call Graph Generator.

3. Uji *user experience*

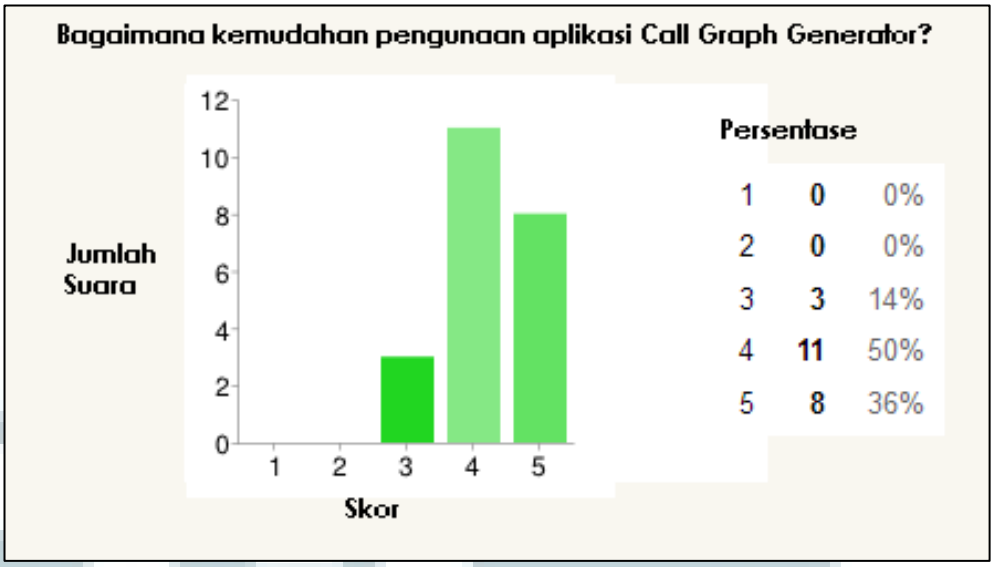
Diagram berikut menunjukkan tingkat kemudahan dalam membaca alur program dengan bantuan aplikasi *Call Graph Generator* dibandingkan dengan secara manual. Skor 5 (lima) berarti sangat mudah dan skor 1 (satu) berarti sangat sulit.




Gambar 4.26 Diagram tingkat kemudahan dalam membaca alur program dengan *Call Graph Generator*.

Jika dilihat dari rata-rata skor respon yang didapat, yaitu 4,13 dapat disimpulkan bahwa dengan adanya bantuan *call graph*, sebagian besar responden dapat membaca alur program dengan lebih mudah.

Diagram berikut menunjukkan tingkat kemudahan dalam menggunakan aplikasi Call Graph Generator. Skor 5 (lima) berarti sangat mudah dan skor 1 (satu) berarti sangat sulit.

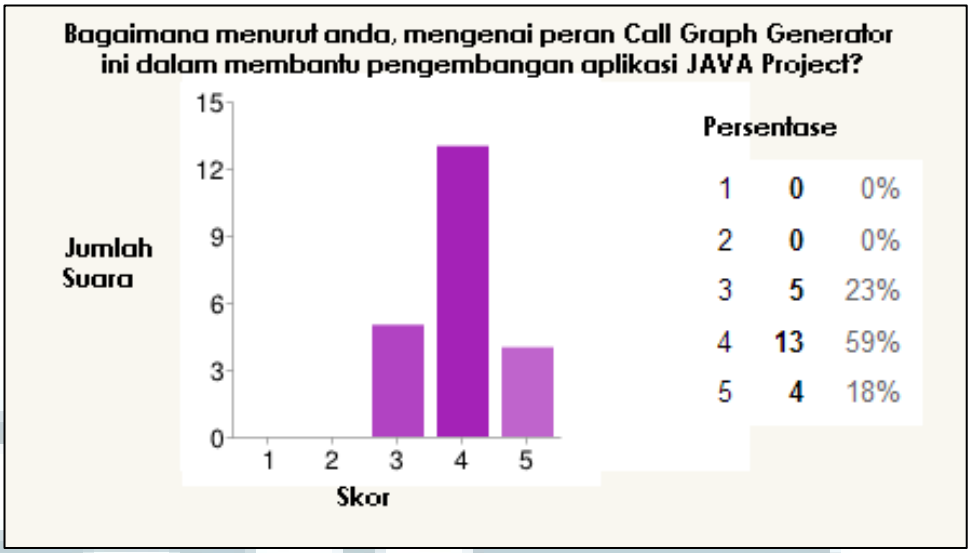



Gambar 4.27 Diagram tingkat kemudahan dalam menggunakan aplikasi *Call Graph Generator*.

Setelah dirata-rata, skor yang didapat pada tingkat kemudahan dalam menggunakan aplikasi adalah 4,2. Dengan skor yang didapat, bisa disimpulkan bahwa desain antarmuka aplikasi Call Graph Generator sudah baik, dimana tidak membingungkan para pengguna.

Berdasarkan grafik yang ditunjukkan pada Gambar 4.28, rata-rata skor yang didapat dalam pertanyaan ini adalah 3,95. Dapat disimpulkan bahwa pengaruh dari aplikasi call graph generator ini tergolong membantu para responden dalam mengembangkan aplikasi mereka.

Diagram berikut menunjukkan tingkat bagaimana peran call graph generator dalam membantu pengembangan aplikasi JAVA. Skor 5 (lima) berarti sangat membantu dan skor 1 (satu) berarti sangat tidak membantu.




Gambar 4.28 Diagram peran aplikasi *Call Graph Generator* dalam membantu pengembangan JAVA *Project.*