BAB II

TINJAUAN PUSTAKA

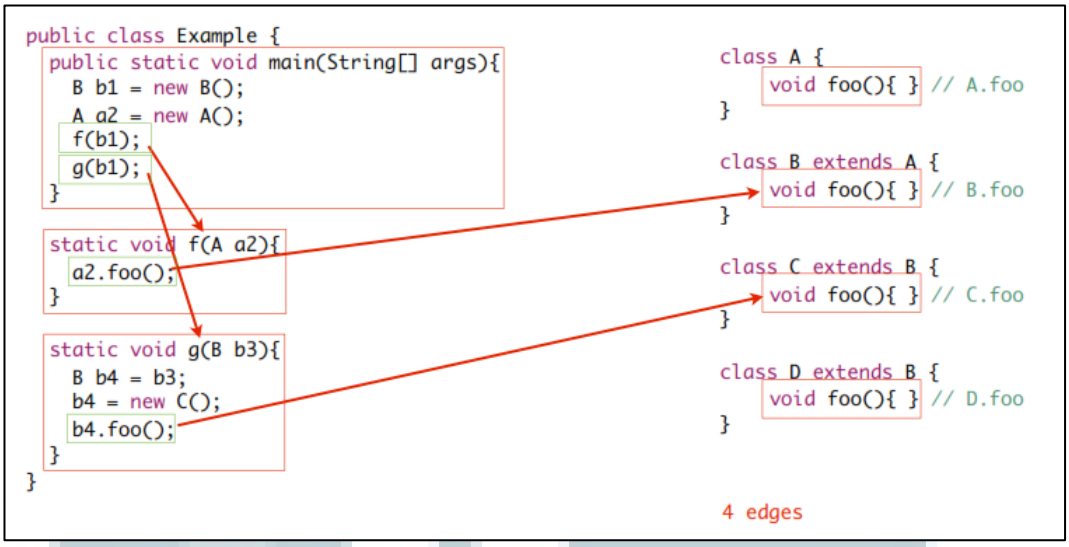
2.1 Call Graph

Call graph adalah sebuah grafik yang menggambarkan relasi antara *sub-routines* dalam program komputer. Setiap *node*-nya melambangkan sebuah *procedure* dan setiap *edge* menggambarkan pemanggilan *procedure*. *Call graph* merupakan hasil analisis dari sebuah program yang dapat digunakan untuk memahami sebuah program, atau sebagai basis/dasar untuk analisis lebih lanjut. Contoh aplikasi yang paling sederhana adalah untuk mengetahui *procedure* yang tidak pernah digunakan.

Ada 2 jenis *Call Graph,* yaitu *dynamic* dan *static*. *Dynamic call graph* dibuat dalam 1 kali program dijalankan, dengan cara merekam *procedure* yang digunakan. Sedangkan *static call graph* menampilkan semua kemungkinan dari program (Grove, DeFouw, Dean, & Chambers, 1997).

Berdasarkan *source code* yang ditampilkan pada Gambar 2.1, dapat disimpulkan bahwa alur pemanggilan fungsinya adalah sebagai berikut

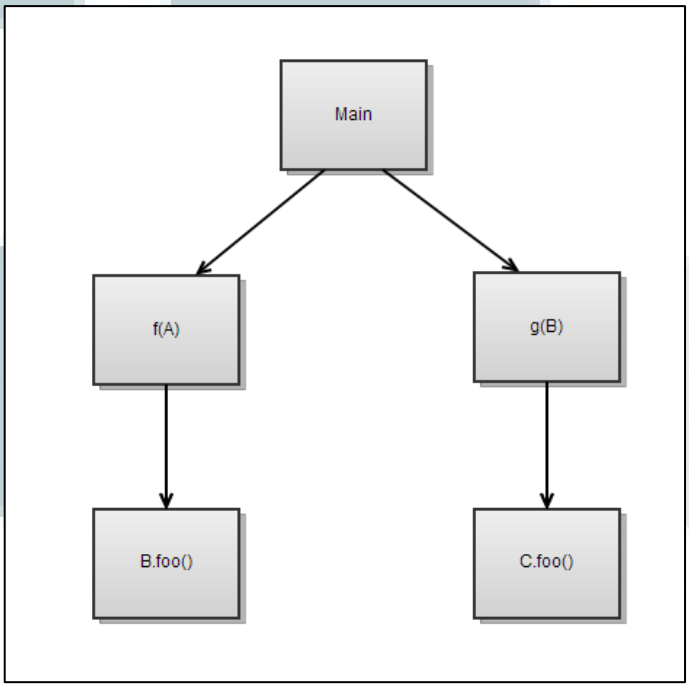
- Fungsi *Main* memanggil fungsi *f.*
- Berdasarkan *parameter* yang didapat dari *Main*, fungsi *f* memanggil fungsi *B.foo*.
- Fungsi *Main* memanggil fungsi *g*
- Fungsi g memanggil fungsi *C.foo*.



Gambar 2.1 Contoh *source code.*

Setelah menentukan alur pemanggilan, Call Graph pun bisa dibuat berdasarkan :

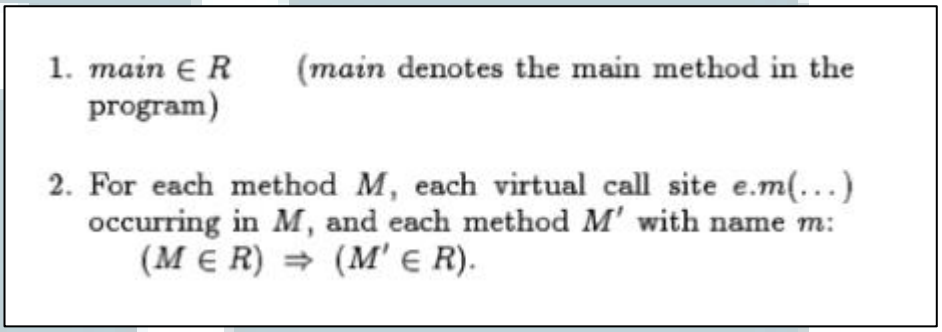
- Setiap *node* untuk setiap *method* /fungsi dalam program
- Untuk setiap pemanggilan, terdapat *edge* dari antara *node* pemanggil dan yang dipanggil.



Gambar 2.2 Contoh *call graph.*

2.2 Reachability Analysis (RA)

Algoritma *Reachability Analysis* (RA) adalah algoritma pertama dan yang paling sederhana dalam konstruksi *Call Graph.* Di balik kesederhanaan-nya, algoritma ini jarang diimplementasikan karena hasilnya yang kurang akurat dalam menunjukkan relasi antar *method.* Berikut adalah langkah-langkah dari algoritma RA.


1. $main \in R$ (*main* denotes the main method in the program)

2. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, and each method M' with name m:
$(M \in R) \Rightarrow (M' \in R)$.

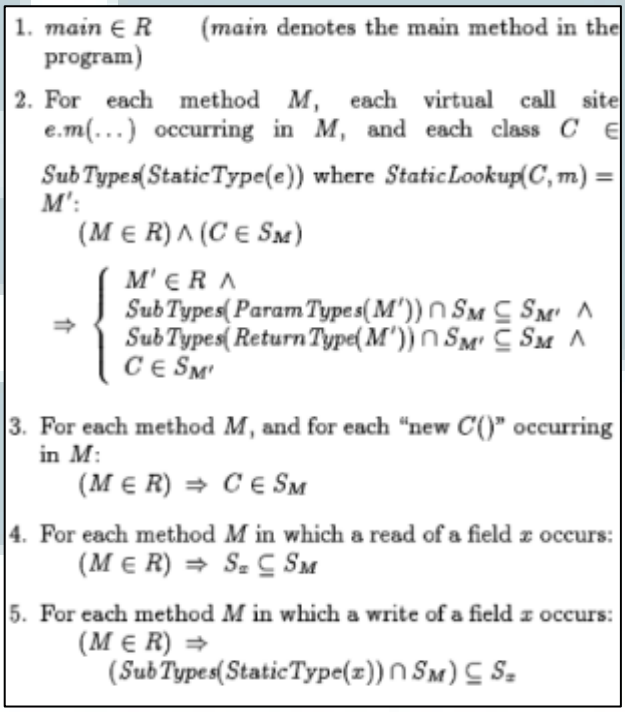
Gambar 2.3 Langkah-langkah algoritma RA (Srivastava, 1992).

Algoritma ini juga sering disebut dengan *name-based resolution*, karena hanya berdasarkan nama method dalam menentukan relasi antar *method.* Dikarenakan algoritma ini sama sekali tidak melihat *return type* atau *parameter* dari sebuah *method*, algoritma ini bekerja dengan cepat, hemat memori, namun *output* yang dihasilkan kurang akurat. Tetapi algoritma ini terkadang dipakai pada aplikasi *software* untuk menghilangkan *method* yang tidak terpakai dalam waktu yang cepat (Jahromi & Honar, 2010).

2.3 Separate Type Analysis (XTA)

XTA dikenal sebagai algoritma konstruksi *call graph* yang paling terfokus. Algoritma ini diperkenalkan oleh Frank Tip dan Jens Palsberg pada

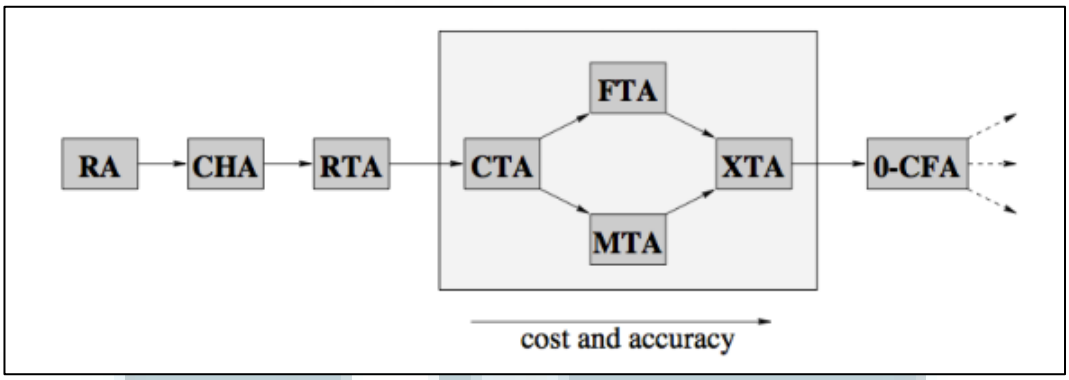
tahun 2000. Algoritma ini meningkatkan algoritma sebelumnya, yaitu *Rapid Type Analysis* (RTA) yang memiliki kecepatan yang tinggi tetapi ketepatannya masih kurang dan sebaliknya, algoritma 0-CFA yang memiliki ketepatan yang tinggi, namun skalabilitasnya masih dipertanyakan. Berdasarkan sifat dan kelebihan dari kedua algoritma ini, dibuatlah XTA yang memiliki keseimbangan antara kecepatan dan ketepatan dalam men-*generate call graph*. (Tip & Palsberg, 2000).



Gambar 2.4 Langkah-langkah algoritma XTA (Tip & Palsberg, 2000).

Algoritma untuk membuat *Call Graph* sudah mulai diteliti sejak tahun 1990-an. Berawal dari algoritma pembentukan *Call Graph* yang paling sederhana yaitu *Reachability Analysis* (RA), *Class Hierarchy Analysis* (CHA), *Rapid Type Analysis* (RTA), *Control Flow Analysis* (0-CFA), XTA dan lain-lain. Dimana algoritma-algoritma ini bertujuan untuk menghasilkan *call graph* yang akurat dan

memiliki skalabilitas yang tinggi pula (untuk menganalisis *source code* yang berskala besar).

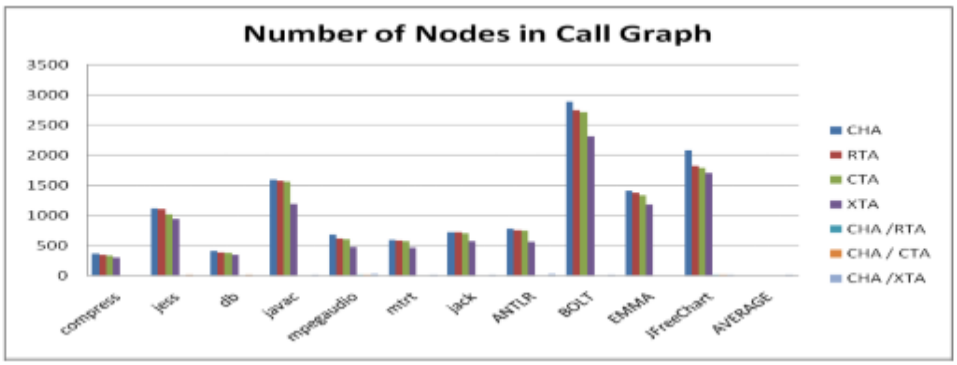


Gambar 2.5 Algoritma-algoritma pembentukan *call graph* dilihat dari biaya dan ketepatan (Tip & Palsberg, 2000).

Dalam penelitian mengenai algoritma pembentukan call graph yang sudah dilakukan sebelumnya, Sajad Ahmad Bhat dan Dr. Jatender Singh dalam *paper*-nya melakukan komparasi dengan *parameter-parameter* sebagai berikut

- Jumlah *Node*

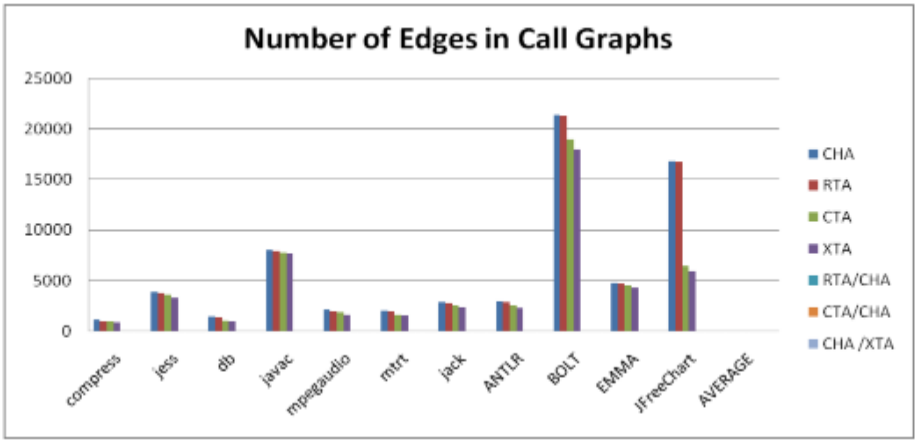
  Semakin sedikit jumlah *node* yang dihasilkan dalam pembentukan call graph, menunjukkan ketepatan dari *call graph* yang dihasilkan.



Gambar 2.6 Grafik algoritma pembentukan *Call Graph* berdasarkan jumlah *node* (Bhat & Singh, 2012).

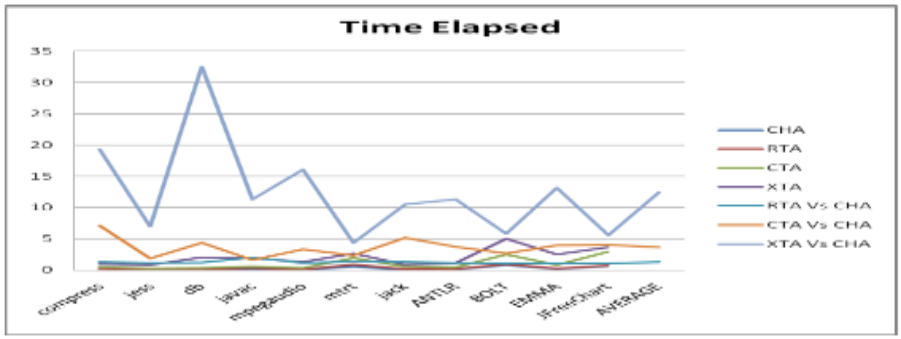
- Jumlah *Edge*

  Sama halnya dengan jumlah *node*, semakin sedikit *edge* yang dihasilkan pada *call graph*, menunjukkan seberapa tepat *call graph* yang dihasilkan.



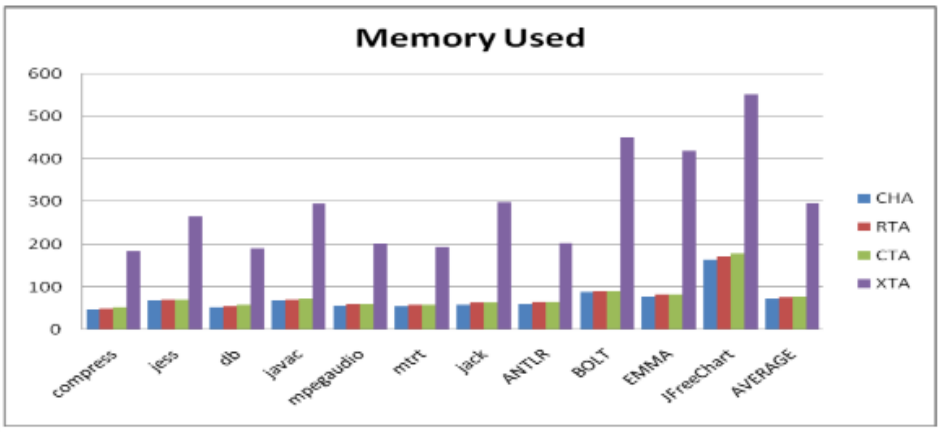
Gambar 2.7 Grafik algoritma pembentukan *Call Graph* berdasarkan jumlah *edge* (Bhat & Singh, 2012).

- Waktu yang terpakai



Gambar 2.8 Grafik algoritma pembentukan *Call Graph* berdasarkan waktu yang diperlukan (Bhat & Singh, 2012).

- Memori yang terpakai



Gambar 2.9 Grafik algoritma pembentukan *Call Graph* berdasarkan jumlah memori yang terpakai (Bhat & Singh, 2012).

2.4 Bahasa Pemrograman JAVA

JAVA merupakan bahasa pemrograman yang berfungsi umum, *object-oriented* dan berbasis pada *class-class*. Diperkenalkan oleh James Gosling, Mike Sheridan dan Patrick Naughton pada Juni 1991 (Oracle). Karakteristik bahasa pemrograman JAVA yang paling utama adalah "*Write Once, Run Everywhere*" dimana program komputer yang ditulis menggunakan bahasa pemrograman ini bisa berjalan sama pada setiap *platform* dengan *virtual machine*, selain itu JAVA juga menggunakan *Automatic Garbage Collector* untuk me-*manage* pemakaian memori dari *object*-nya (Java).

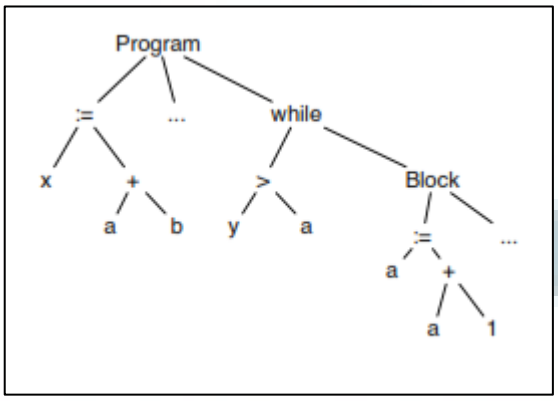
Bahasa pemrograman JAVA merupakan salah satu bahasa pemrograman terpopuler digunakan oleh programmer-programmer di seluruh dunia. Pada tahun 2009, NumberOf.net mencatat bahwa ada 9.007.346 developer JAVA di seluruh dunia (NumberOf.net, 2009). Angka ini didapat setelah menganalisis berbagai

sumber yaitu pasar pekerjaan, forum-forum, jumlah pengunduhan piranti lunak produk-produk JAVA yang salah satunya adalah Eclipse (Eclipse Juno diunduh sebanyak 8 juta kali).

Program yang ditulis menggunakan JAVA memiliki reputasi lebih lambat dan membutuhkan memori yang lebih besar dibandingkan dengan program yang ditulis menggunakan C++. Pada desember 2012, microbenchmarks menunjukkan bahwa JAVA 7 lebih lambat 44% dibandingkan C++ (Jelovic).

2.5 Abstact Syntax Tree

Abstract Syntax Tree (AST) adalah sebuah *tree* yang merepresentasikan struktur *syntax* dari sebuah *source code* yang ditulis dalam sebuah bahasa pemrograman tertentu. Setiap *node* dari *tree* melambangkan sebuah konstruksi dalam *source code*. AST merupakan struktur data yang sering digunakan dalam berbagai macam *compiler*, karena AST bisa menjadi sebuah *intermediate representation* (IR) dari program yang dibutuhkan oleh *compiler* dan sangat berpengaruh dalam menentukan *output* dari *compiler* tersebut (Howe, 1994).



Gambar 2.10 Contoh Abstact Syntax Tree (AST) (University of Maryland, 2011).

AST sangat sering digunakan dalam analisis semantik, dimana *compiler* memeriksa penggunaan elemen dari program dan bahasa pemrograman. Selain itu, selama analisis semantik, *compiler* akan menghasilkan tabel simbol berdasarkan AST ini. Alur dari tree ini dapat menentukan benar atau tidaknya program (Jones).

2.6 Eclipse

Eclipse adalah sebuah IDE (Integrated Development Environment) untuk mengembangkan perangkat lunak dan dapat dijalankan di semua *platform*. Eclipse dikembangkan oleh IBM pada 5 November 2001, sebagai pengganti dari program "Visual Age for JAVA". Setelah itu, Eclipse Foundation mengambil alih untuk pengembangan Eclipse lebih lanjut dan pengaturan organisasinya (Eclipse).

Eclipse sendiri bersifat *multi-platform* yang dimana bisa dijalankan di berbagai sistem operasi. Eclipse dikembangkan dengan bahasa pemrograman JAVA, akan tetapi juga tetap mendukung pengembangan aplikasi berbasis bahasa pemrograman lainnya. Selain sebagai IDE untuk pengembangan aplikasi, Eclipse juga bisa digunakan untuk aktivitas dalam siklus pengembangan perangkat lunak, seperti dokumentasi, test, pengembangan web dan lain sebagainya.

2.7 Eclipse *Plug-In*

Salah satu kelebihan Eclipse sebagai IDE adalah arsitekturnya yang bisa dikembangkan oleh penggunanya dengan komponen yang dinamanakan *plug-in*. Eclipse *Plug-in* merupakan sebuah komponen yang bisa memberikan berbagai

macam *service* terkait dengan *workbench* dari Eclipse. Sistem *plug-in* Eclipse ini dibuat berdasarkan Equinox, yang dimana merupakan implementasi dari framework OSGi. Infrastruktur dan modul *service* pada Eclipse ini bisa dibuat berupa *bundle* dan bisa digunakan kembali dan di-integrasikan (Balsiger, 2010).

Dalam pengembangan *project* Eclipse *plug-in*, terdapat beberapa hal-hal yang menjadi dasar pengembangan, yaitu *Manifest, Activator.java, Action.java*

- **Manifest**

  Berisi informasi yang akan digunakan oleh Eclipse untuk mengintegrasikan plug-in kedalam *framework*. Di dalamnya terdapat lagi 3 hal dasar, yaitu *general information* yang berisikan informasi dasar dari plug in (seperti nama, versi, nama package), *action set* yang berisi *action* apa saja yang akan dijalankan oleh *plug in* dan *plug in dependencies* yang berisi tentang *package* apa saja yang dibutuhkan untuk menjalankan *plug in*.

- **Activator.java**

  Merupakan *class* yang meng-*extend class Abstract* bernama AbstractUIPlugin yang bertanggung jawab atas *life cycle* dari keseluruhan *plug in*, dari sejak saat *plug in* dijalankan hingga berhenti.

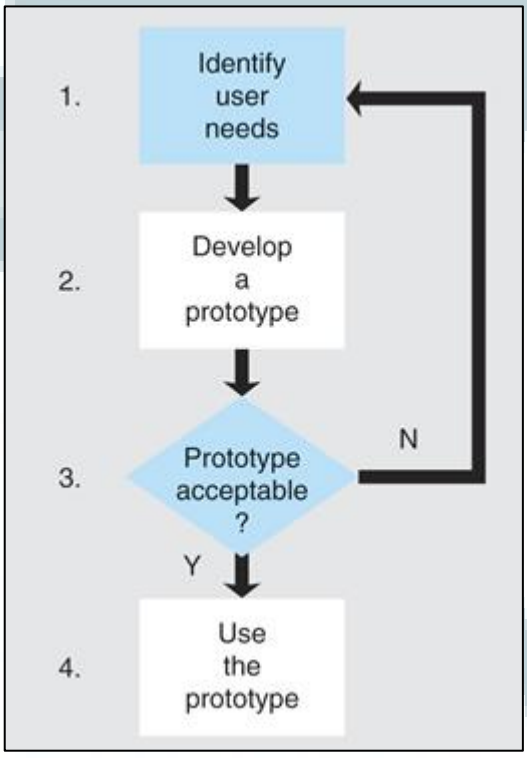
- **Action.java**

  Merupakan *class* yang yang menjalankan *action* yang telah dideklarasikan pada *file manifest*. *Class Action* ini mengimplementasikan *interface* dari IWorkbenchWindowActionDelegate yang berguna supaya Eclipse bisa

menggunakan *proxy* untuk menentukan, apakah *plug in* ini butuh di-*load* atau tidak (untuk optimalisasi memori dan performa) (IBM, 2002).

2.8 Metode pengembangan Prototyping

Prototyping adalah proses pembuatan model sederhana dari piranti lunak yang mengijinkan pengguna memiliki gambaran dasar tentang program serta melakukan pengujian awal. *Prototyping* memberikan fasilitas bagi pengembang dan pemakai untuk saling berinteraksi selama proses pembuatan, sehingga pengembang dapat dengan mudah memodelkan perangkat lunak yang akan dibuat (McLeod & P.Schell, 2001).



Gambar 2.11 *Flowchart* metode pengembangan *prototyping* (McLeod & P.Schell, 2001).

Model prototyping secara umum memiliki 3 proses (McLeod & P.Schell, 2001), yaitu

1. Pengumpulan kebutuhan

   Dalam proses ini, developer dan pengguna bertemu dan menentukan tujuan umum, kebutuhan yand diketahui dan gambaran bagian-bagian yang akan dibutuhkan berikutnya

2. Perancangan dan pembuatan prototype

   Perancangan dan pembuatan prototype dilakukan berdasarkan kebutuhan yang sudah didapat.

3. Evaluasi *prototype*

   Prototype di evaluasi, apakah sesuai dengan kebutuhan. Jika tidak, proses kembali ke pengumpulan kebutuhan. Sebaliknya jika sudah sesuai, prototype akan dipakai.