Framework Deteksi Bad Smell Code Semi Otomatis untuk Pemrograman Tim

Tugas Akhir Mata Kuliah Keamanan Perangkat Lunak (EL5215)
Yusfia Hafid Aristyagama
(23214355)
Magister Teknik Elektro TMDG'09
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2016

ABSTRAK

Kualitas dari *software* sangat tergantung pada kualitas desain rancangan perangkat lunak tersebut. Untuk meningkatkan kualitas perangkat lunak tersebut, maka perbedaan antara kondisi status awal dan status berikutnya harus terukur. Namun, melakukan penilaian desain perangkat lunak merupakan hal yang sangat sulit untuk dilakukan. Beberapa studi telah melakukan penelitian yang berfokus pada penilaian desain perangkat lunak secara otomatis. *Bad smell* merupakan salah satu faktor yang sulit untuk diidentifikasi karena perbedaan pengalaman dari setiap *programmer*. Hal tersebut terjadi karena tidak tersedianya standarisasi atau parameter yang jelas untuk melakukan klasifikasi *bad smell* yang jelas. Meskipun pemrograman *software* tersebut dilakukan dalam tim, pandangan setiap *programmer* terhadap desain perangkat lunak seperti *bad smell* selalu berbeda-beda. Padahal, kualitas desain perangkat lunak yang buruk, sangat berpotensi untuk menimbulkan ancaman bagi keberlangsungan proyek di masa depan.

Dengan permasalahan tersebut, makalah ini bermaksud untuk menyediakan sebuah rancangan desain *framework* deteksi *bad smell code* semi otomatis untuk menangani permasalahan standarisasi *bad smell code* dalam pemrograman tim. *Expert System* dimanfaatkan sebagai model dari *knowledge base*. Perancangan stuktur *knowledge base* dilakukan dengan menggunakan analisis entropi dan gain terhadap *software metrics* terkait.

Kata kunci : *bad smell*, deteksi, semi otomatis, *framework*, tim.

1. PENDAHULUAN

Teknologi selalu berubah dan terus berkembang seiring dengan kebutuhan manusia yang semakin bervariasi. Hal tersebut berdampak terhadap pengembangan perangkat lunak maupun perangkat keras. Dalam pengembangan perangkat lunak, semakin banyak spesifikasi yang diminta oleh *client*, maka desain aplikasi akan menjadi semakin kompleks. Menurut literatur [1], ketika sebuah perangkat lunak dikembangkan, dimodifikasi, dan diadaptasikan terhadap spesifikasi baru, *source code* yang dihasilkan akan semakin kompleks dan rumit. Probabilitas terjadinya eror perangkat lunak yang mengakibatkan celah keamanan perangkat lunak menjadi lebih besar seiring dengan terjadinya peningkatan jumlah *source code* dalam rancangan perangkat lunak tersebut.

Dalam pengembangan *software* skala besar, kesesuaian terhadap standar koding merupakan hal yang sangat penting dalam penilaian kualitas *source code*. Jika *source code* tidak diorganisir secara terstruktur dan baik, maka hal tersebut akan mempersulit pengembangan, integrasi, dan perawatan *software* di masa yang akan datang [2]. Dalam literatur [3] disebutkan bahwa kualitas dari *software* sangat tergantung pada kualitas desain rancangan *software* tersebut. Desain rancangan yang buruk akan sulit untuk diubah karena sulit untuk menemukan tempat dimana perubahan tersebut dibutuhkan [4]. Jika tempat tersebut sulit untuk ditemukan, maka besar kemungkinan dimana *programmer* akan melakukan kesalahan yang menimbulkan *bugs*. Oleh karena itu melakukan *refactoring* secara berkala pada setiap kondisi dimana *refactoring* dibutuhkan akan mengurangi potensi munculnya desain rancangan *software* yang buruk dan mengurangi risiko munculnya *bugs*. Salah satu kondisi dimana *refactoring* diperlukan adalah ketika terdeteksi *bad smell* di dalam *source code*. Fowler dalam literatur [4] menyatakan bahwa dengan melakukan *refactoring* terhadap *bad smell* akan menambah kualitas desain *software*, membuat *source code* lebih mudah dipahami, memudahkan pencarian *bugs*, dan membantu melakukan pemrograman secara lebih cepat. Proses deteksi *bad smell* adalah operasi awal yang sangat penting untuk dilakukan sebelum melakukan *refactoring*.

Menurut literatur [2], dijelaskan bahwa kualitas *source code* dapat ditingkatkan dengan memastikan bahwa *source code* tersebut telah sesuai dengan standar koding. Standar tersebut harus dapat memastikan bahwa perubahan kualitas *source code* yang dihasilkan adalah perubahan yang terukur. Namun, melakukan pengukuran *bad smell* pada *source code* merupakan hal yang sangat sulit untuk dilakukan karena setiap individu (*programmer*) memiliki persepsi yang berbeda tentang *bad smell*. Menurut literatur [2], menerbitkan buku pedoman aturan koding saja belum cukup untuk menyelesaikan masalah standardisasi koding dalam sebuah *software house*. Jika pengembang ataupun *programmer* tidak dapat melihat esensi dari aturan standar koding, tidak percaya bahwa aturan tersebut akan berguna suatu saat, dan merasa dibatasi dengan adanya aturan tersebut, maka pengembang ataupun *programmer* lebih cenderung untuk mengabaikan aturan tersebut. Seperti yang telah disimpulkan dalam survei evaluasi subjektif terhadap pengembang *software* pada literatur [3], variasi persepsi tersebut tergantung pada latar belakang setiap *programmer* seperti peran, pengetahuan, dan pengalaman kerja. Dalam *software house*, standardisasi koding merupakan hal yang perlu untuk dilakukan untuk memastikan kualitas *source code*. Namun, pengalaman perbedaan persepsi *programmer* dalam kasus *bad smell* ini adalah sebuah kendala untuk melakukan standarisasi tersebut.

Sebuah *framework* untuk melakukan deteksi *bad smell* dalam pemrograman berkelompok (tim) diperlukan. Dengan kendala yang telah disampaikan sebelumnya, maka tujuan dari diusulkannya *framework* tersebut dapat dirumuskan sebagai berikut:

- Media pembelajaran *tim programmer* untuk menyamakan persepsi tentang *bad smell*.
- Mendeteksi keberadaan *bad smell* di dalam *source code*.
- Mengevaluasi, mengurangi, ataupun menambahkan prasyarat klasifikasi *bad smell* berdasarkan pengalaman *programmer* tertentu.

Untuk memenuhi tujuan tersebut maka perlu dibangun sebuah *framework* dengan kriteria sebagai berikut:

- *Reasonable* dan *explainable*: yaitu kondisi dimana *framework* tersebut mampu memberikan penjelasan dan sebab-sebab yang rasional ketika *framework* tersebut mengklasifikasikan bagian dari *source code* sebagai *bad smell*. Dengan demikian, programmer pemula dapat mempelajari karakteristik dari *bad smell* berdasarkan penjelasan yang disampaikan oleh *framework*.
- *Traceable*: yaitu kondisi dimana *framework* tersebut mampu untuk mendeteksi letak *bad smell* di dalam *source code*.
- *Adaptive*: yaitu kondisi dimana sebuah *framework* mampu beradaptasi terhadap perubahan *knowledge base* berdasarkan pengetahuan dan pengalaman *programmer*. Dengan demikian *framework* yang dibangun mampu untuk belajar dengan cara mengevaluasi, mengurangi, ataupun menambahkan pengetahuan berdasarkan fitur *source code* tertentu yang dipilih untuk mengidentifikasi parameter *bad smell*. Nilai parameter tersebut akan dijadikan sebagai standar untuk mengklasifikasikan dan mengidentifikasi keberadaan *bad smell* dalam pemrograman berkelompok.

Dalam makalah ini diusulkan sebuah *framework* untuk menangani deteksi *bad smell* secara semi-otomatis dalam pemrograman tim. Pada bagian ke-1, dijelaskan tentang latar belakang pengusulan *framework*. Jenis-jenis *bad smell* akan dibahas pada bagian ke-2. Pekerjaan sebelumnya yang menjadi inspirasi *framework* dijelaskan pada bagian ke-3. Sementara, untuk desain *framework* yang diusulkan akan dibahas pada bagian ke-4. Pada bagian ke-5, disampaikan kesimpulan dari pembahasan yang telah disampaikan pada bagian-bagian sebelumnya.

2. BAD SMELL

Istilah *bad smell* dikenalkan oleh Fowler pada literatur [4]. Literatur [5] menyebut hal ini dengan istilah *design flaws*. Dengan esensi yang sama seperti *bad smell*, terdapat istilah lain yang dikenal sebagai *anti-pattern*. Meskipun secara esensi sama, namun struktur dan model klasifikasi yang diperkenalkan pada istilah *anti-pattern* ini berbeda. Berdasarkan literatur [4] dan [6] dapat disimpulkan bahwa istilah *bad smell* ini merujuk kepada pola struktur bagian *source code* dari *software* tertentu yang memiliki desain buruk, sehingga menjadi ancaman yang berpotensi menimbulkan *bugs*, eror, ataupun celah keamanan di masa yang akan datang. Keberadaan *bad smell* dalam sebuah *source code* merupakan sebuah indikasi bahwa pengembang atau *programmer software* tersebut perlu untuk melakukan *refactoring* terhadap *source code* dari *software* tersebut. Caranya adalah dengan melakukan transformasi untuk mengubah struktur internal *software* yang terdeteksi sebagai *bad smell* tanpa mengubah *behavior* dari program. Beberapa literatur menyebutkan jenis-jenis *bad smell* (termasuk *anti-pattern*) yang diantaranya adalah sebagai berikut:

- *Duplicated Source Code*
  Dalam literatur [4], *Duplicated Source Code* merupakan salah satu bentuk *bad smell* yang memiliki ciri-ciri yaitu terdapatnya beberapa struktur *source code* yang sama di beberapa tempat. Contoh sederhananya adalah ketika terdapat dua struktur *method* yang sama dalam satu *class*. Contoh lainnya semisal terdapat dua atau lebih struktur ekspresi yang sama di dalam dua atau lebih *child class* dengan *parent class* yang sama.

- *Long Method*
  Berdasarkan penjelasan yang disampaikan oleh [4] dan [6], dapat disimpulkan bahwa *bad smell* yang tergolong sebagai *Long Method* adalah *method* yang memiliki terlalu banyak baris kode untuk dieksekusi. Akibatnya *method* tersebut menjadi lebih sulit untuk dipahami. Dalam literatur [4], Fowler menambahkan ciri-ciri fungsionalitas dan kompleksitas operasi yang banyak. Namun literatur [6] mendefinisikan ulang *Long Method*, dengan menghilangkan syarat kompleksitas operasi. Tujuannya adalah agar lebih mudah untuk membedakan antara *Long Method* dengan *God Method.* Dalam kasus ini, maka definisi yang diajukan oleh [6] akan digunakan dalam pembahasan selanjutnya.
- *God Method*
  Menurut literatur [6], *method* yang disebut sebagai *God Method* adalah *method* yang melakukan terlalu banyak fungsional. Jika dibandingkan, maka perbedaan antara *Long Method* dengan *God Method* terletak pada kata kunci jumlah baris dan kompleksitas. *God Method* lebih cenderung identik dengan kompleksitas, sementara *Long Method* lebih cenderung identik pada jumlah baris. Hal tersebut ditandai dengan adanya logika bisnis yang terlalu banyak dilakukan dalam *method* tersebut.
- *Brain Method*
  Berdasarkan hasil pembelajaran pada literatur [7], ditemukan sebuah istilah baru yaitu *Brain Method*. *Brain Method* merupakan bagian yang tidak terpisahkan dari *Brain Class*. *Brain Method* memiliki kompleksitas operasi yang tinggi dan menjadi pusat kecerdasan dari sistem. Apabila terdapat lebih dari atau sama dengan satu *Brain Method* dalam sebuah *class*, maka *class* tersebut terindikasi sebagai *Brain Class*. Dalam hal ciri-ciri, tidak ada perbedaan jauh diantara *Brain Method* dengan *God Method*. *Brain Method* dan *God Method* sama-sama menekankan kompleksitas logika dan operasi yang tinggi sebagai fitur utamanya. Perbedaannya adalah *Brain Method* memanfaatkan *software metrics* LOC untuk melakukan pengukuran, sementara *God Method* tidak menekankan LOC sebagai salah satu tolak ukur. Hal tersebut berarti bahwa jumlah baris atau jumlah *statement* yang dieksekusi merupakan pembeda antara *Brain Method* dan *God Method*.
- *Large Class*
  Literatur [4] menyebutkan bahwa ciri-ciri *Large Class* adalah ketika terdapat sebuah *class* yang melakukan terlalu banyak pekerjaan. Hal tersebut ditandai dengan terlalu banyaknya variabel instansiasi. Apabila dalam sebuah *class* terdapat banyak variabel instansiasi, pada *class* tersebut umumnya juga akan terdapat *Duplicated Source Code*.
- *God Class (Blob)*
  God Class merupakan salah satu bentuk dari *Large Class*. Apabila sebuah *class* terindikasi sebagai *God Class*, maka *class* tersebut juga akan dinyatakan sebagai *large class*. Menurut literatur [7], sebuah *class* disebut sebagai *God Class* apabila instansiasi dari *class* tersebut melakukan banyak pekerjaan, hanya mendelegasikan sedikit dari beberapa detail tersebut ke *class* yang lain, dan menggunakan data terlalu banyak dari *class* lain. Dengan kata lain, *god class* ini memiliki kompleksitas operasi yang tinggi, memiliki kohesi *inner-class* yang rendah, dan terlalu banyak menggunakan data dari *class* lain. Hal ini mengakibatkan *class* tersebut seolah-olah menjadi pusat kecerdasan dari sistem. Apabila dilakukan *refactoring*, maka pada umumnya *god class* ini paling banyak mengalami perubahan jika dibandingkan dengan *class* yang lain. Dalam literatur [8], disebutkan bahwa *God Class* merupakan salah satu jenis *anti-pattern*.
- *Brain Class*
  Selain *God Class*, *Brain Class* juga merupakan salah satu bentuk dari *Large Class*. *Brain class* juga memiliki kompleksitas operasi yang tinggi sehingga seolah-olah menjadi pusat kecerdasan dari sistem. Perbedaannya dengan *God Class* adalah bahwa *Brain Class* tidak banyak mengakses data dari *class* lain dan sedikit lebih kohesif apabila dibandingkan dengan G*od Class* [7].
- *Spaghetti Code*
  Spaghetti Code disebutkan sebagai salah satu jenis dari *anti-pattern* di dalam literatur [8]. *Class* yang tergolong sebagai *Spaghetti Code* tidak mendukung mekanisme pemrograman berorientasi obyek seperti *polimorphism* dan *inheritance*. Ciri-ciri dari *Spaghetti Code* adalah terdapatnya *class* dengan sedikit atau tanpa struktur namun disertai dengan pendeklarasian *method* yang terlalu panjang tanpa adanya parameter. *Class* tersebut menggunakan variabel global untuk melakukan pemrosesan. Pada umumnya baik nama *class* maupun *method* yang digunakan seolah-olah bersifat prosedural.
- *Long Parameter List*
  Long Parameter List merupakan salah satu bentuk dari *bad smell* yang disebutkan dalam literatur [4]. Pada pemrograman berorientasi obyek seharusnya parameter yang digunakan untuk *method*, fungsi, ataupun prosedur cenderung lebih pendek jika dibandingkan dengan program sekuensial. Hal tersebut terjadi karena parameter yang seharusnya panjang dapat dapat disederhanakan dengan melakukan *passing* variable berupa obyek dimana obyek tersebut memiliki atribut yang dibutuhkan oleh method lain. Dengan demikian *method* seharusnya menjadi lebih mudah dipahami. *Long Parameter List*

mempunyai ciri-ciri spesifik, yaitu ketika sebuah *method* ataupun fungsi memiliki banyak parameter masukan. Umumnya hal tersebut akan membingungkan karena parameter-parameter tersebut lebih sulit untuk didefinisikan dan sulit untuk digunakan.

- *Divergent Change*
  Berdasarkan literatur [4], *Divergent Change* merupakan salah satu bentuk *bad smell* yang mengharuskan banyak perubahan pada sebuah *class* akibat dari sebuah perubahan untuk menangani variasi data tertentu. Berdasarkan penjelasan tersebut, berarti *Divergent Change* ini merupakan bentuk dari *bad smell* yang terjadi akibat masalah kohesifitas dalam *class*.
- *Shotgun Surgery*
  Berdasarkan literatur [4], *Shotgun Surgery* merupakan bentuk *bad smell* dimana ketika setiap kali perubahan dilakukan maka dibutuhkan perubahan-perubahan kecil juga pada *class* lain yang berhubungan sebagai dampaknya.
- *Feature Envy*
  Feature Envy merupakan salah satu bentuk *bad smell* yang terjadi akibat tingginya tingkat *coupling* antar *class*. Semakin tinggi tingkat *coupling* antar *class* maka akan semakin banyak *class* yang terpengaruh apabila dilakukan perubahan dalam *source code* [9]. *Feature Envy* terjadi akibat adanya sebuah *method* ataupun *class member* yang cenderung lebih tertarik terhadap *class* lain dibandingkan dengan *class* dimana *method* ataupun *class member* tersebut didefinisikan [4]. Pada umumnya hal tersebut merupakan dampak yang terjadi karena kesalahan penempatan *class member*.
- *Data Clumps*
  Menurut [4], ketika ditemukan dua atau lebih data maupun primitif yang selalu dideklarasikan bersama-sama dan saling bergantung maka hal tersebut merupakan sebuah indikasi bahwa dalam *source code* tersebut terdapat *Data Clumps*. Contohnya ketika ditemukan keberadaan pasangan variabel *start* dan *end* disetiap *class* ataupun parameter *method*. *Data Clumps* pada umumnya berupa pasangan nilai primitif yang seharusnya masih dapat diubah kembali strukturnya dengan menjadikannya sebagai sebuah obyek.
- *Primitive Obsession*
  Berdasarkan literatur [4],*Primitive Obsession* merupakan salah satu *bad smell* yang ditandai dengan preferensi penggunaan variabel tipe primitif untuk merepresentasikan tugas yang sederhana dibandingkan dengan variabel objek. Misalnya menggunakan *string* untuk kode pos atau *integer* untuk mata uang.
- *Switch Statements*
  Di dalam literatur [4], penggunaan *switch* dan *case* merupakan ciri-ciri dari *Switch Statement*. Dibandingkan dengan pemrograman berorientasi obyek, penggunaan *switch* lebih cenderung menunjukkan ciri-ciri dari pemrograman prosedural. Efek yang ditimbulkan adalah kemungkinan duplikasi *switch statement* yang sama dimana duplikasi tersebut tersebar di tempat yang berbeda dalam satu proyek. Hal tersebut akan mempersulit proses *refactoring* saat dibutuhkan. Dibandingkan penggunaan switch, pemrograman berorientasi obyek menawarkan solusi yang lebih elegan yaitu penggunaan konsep *polimorphism*.
- *Parallel Inheritance Hierarchies*
  Bad smell dengan jenis *Parallel Inheritance Hierarchies* ini merupakan salah satu kasus khusus dari *Shotgun Surgery* [4]. Pada *Parallel Inheritance Hierarchies*, setiap sebuah *subclass* dibuat dari *class* tertentu maka *subclass* dari *class* lainnya juga harus dibuat.
- *Lazy Class*
  Berdasarkan literatur [4] dan [10], dapat disimpulkan bahwa *Lazy Class* merupakan sebuah *class* yang seharusnya dieliminasi karena nilai fungsionalitasnya terlalu kecil. *Lazy Class* memiliki ciri-ciri berkebalikan dengan *Large Class*. Namun hal tersebut tidak berarti bahwa untuk melakukan deteksi terhadap *Lazy Class* cukup dengan memberikan parameter deteksi yang berkebalikan dengan *Large Class* [10].
- *Speculative Generality*
  Speculative Generality pada umumnya dapat ditemukan pada *class* ataupun *method* yang digunakan sebagai *test case*. Hal tersebut terjadi karena pengembang mencoba untuk menambahkan fitur-fitur khusus berdasarkan spekulasi pengembang untuk mengantisipasi suatu kondisi di masa yang akan datang. Namun pada kenyataannya fitur tersebut tidak dibutuhkan dan tidak diimplementasikan. Hal tersebut membuat *source code* menjadi lebih sulit untuk dipahami [4]. Berdasarkan penjelasan tersebut ciri-ciri dari *Speculative Generality* adalah terdapatnya *class*, *method*, *field*, ataupun parameter yang tidak digunakan. Contohnya adalah *class interface* atau *abstract* yang tidak memiliki atau memiliki hanya satu *class* turunan saja [11].
- *Temporary Field*
  Literatur [4] menjelaskan bahwa *Temporary Field* adalah kondisi dimana terdapat variabel di dalam sebuah obyek yang hanya akan diinisiasi pada kondisi tertentu. Adanya *Temporary Field* ini akan

membuat sebuah *source code* menjadi lebih sulit untuk dipahami karena seorang pengembang akan berpikir bahwa setiap variabel yang ada di dalam sebuah obyek seharusnya merupakan variabel yang dibutuhkan oleh obyek tersebut. Namun pada kenyataannya *Temporary Field* tidak sepenting yang diduga sehingga menyebabkan kebingungan bagi pengembang yang berusaha memahami *source code* tersebut.

- *Message Chains*
  Sebuah *bad smell* disebut sebagai *Message Chains* apabila untuk mengakses *data field* di *class* yang lain harus memanggil banyak *getter method* dari beberapa *class* secara sekuensial [11].
- *Middle Man*
  Berdasarkan [4] dan [11], *Middle Man* merupakan sebuah *class* dimana setengah atau lebih dari *method* yang didefinisikannya melakukan tugas dengan cara mendelegasikan tugas tersebut ke *class* yang lain. Di dalam setiap *method* delegator tersebut paling sedikit terdapat sebuah referensi ke *class* yang lain.
- *Alternative Classes with Different Interface*
  Berdasarkan literatur [4], kondisi dimana terdapat beberapa *class* dengan fungsi, konseptual, ataupun struktur yang sama disebut sebagai *Alternative Classes with Different Interface*. Pada umumnya hal tersebut terjadi karena *programmer* atau pengembang *software* tidak mengetahui bahwa *class* dengan fungsionalitas yang sama telah dibuat dan telah ada sebelumnya.
- *Incomplete Library*
  Di dalam sebuah aplikasi besar, penggunaan *library* merupakan hal yang sangat membantu proses pengembangan *software*. Namun dalam kondisi tertentu akan ada kondisi dimana sebuah *library* tidak dapat lagi memenuhi harapan kebutuhan pengguna karena pengarang/pencipta dari *library* tersebut tidak menyediakan fitur yang dibutuhkan atau menolak adanya penambahan fitur. Dalam keadaan tersebut, hal yang perlu dilakukan adalah melakukan *upgrade* dan penambahan fitur baru pada *source code*. Namun, pada umumnya *library* bersifat *read-only* sehingga tidak dapat ditambah lagi fiturnya. Menurut [4], kondisi tersebut merupakan *bad smell* yang dapat dikategorikan sebagai *Incomplete Library*.
- *Data Class*
  Dalam literatur [4] dijelaskan bahwa *Data Class* merupakan sebuah *class* yang memiliki ciri-ciri dimana di dalam *class* tersebut hanya terdiri dari *field* beserta *setter* dan *getter method* untuk setiap *field* pada *class* tersebut. *Class* tersebut hanya berfungsi untuk menampung data dan tidak memiliki fungsionalitas ataupun *behavior* lebih. Hal itu menyebabkan *class* tersebut tidak dapat melakukan operasi pada datanya sendiri secara independen.
- *Refused Bequest*
  Refused Bequest merupakan kondisi dimana konsep pemrograman berorientasi obyek *inheritance* tidak digunakan sebagaimana mestinya. Secara teknis, *subclass* tidak menggunakan fungsionalitas turunan dari *superclass*, sehingga hubungan antara *superclass* dan *subclass* tidak mencerminkan hubungan "is-a" [12]. *Subclass* seharusnya mewarisi *method* dan *data* dari *superclass*, Namun, apabila *subclass* tersebut tidak membutuhkan warisan dari *superclass*, maka dapat disimpulkan bahwa hubungan *sublcass* dan *superclass* tersebut merupakan kesalahan hierarki [4].
- *Comments*
  Menurut literatur [4], *comment* sering digunakan untuk menutupi kelemahan *source code* yang sulit dipahami. Daripada menggunakan terlalu banyak *comment*, sebaiknya menggunakan *comment* seperlunya, mengubah struktur *source code* agar menjadi lebih mudah untuk dipahami, dan memberi nama *class* ataupun *method* yang representatif terhadap tugasnya.

Di antara *bad smell* yang telah disebutkan, masing-masing terbagi menjadi beberapa jenis level berdasarkan ciri-cirinya. Sebuah *bad smell* mungkin berada pada level *package*, *class*, atau *method*. Dalam pembahasan makalah ini, metode deteksi yang diusulkan akan difokuskan pada beberapa *bad smell* pada level *class* dan *method*. Pada level *class*, beberapa *bad smell* yang akan dimodelkan adalah *Large Class*, *Brain Class*, *God Class*, dan *Lazy Class*. Pada level *method* beberapa *bad smell* yang akan dimodelkan adalah *Brain Method*, *God Method*, dan *Long Method*. Proses pemodelan akan melibatkan beberapa *software metrics* yang disebutkan pada literatur [13], [14], dan [15]. Tujuan dari penggunaan *software metrics* tersebut adalah sebagai alat ukur klasifikasi *bad smell*.

3. PEKERJAAN TERKAIT

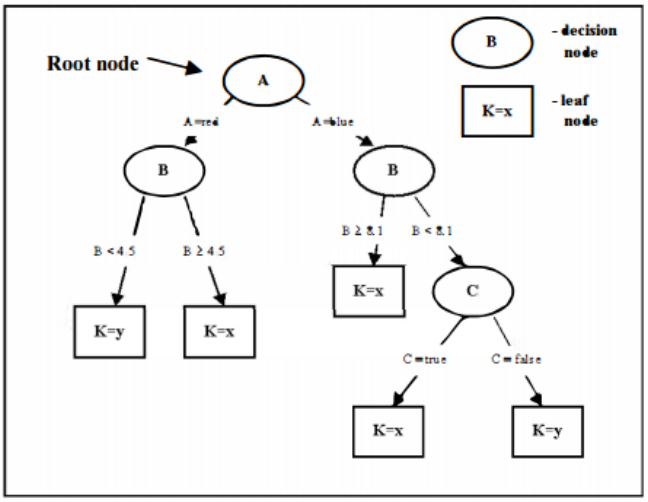
Beberapa metode untuk melakukan deteksi *bad smell* semi-otomatis telah diajukan dalam penelitian sebelumnya seperti *Neural Network* [16], *Decision Tree* [5], dan HIST *(Historical Information for Smell deTection)* [17]. Setiap metoda deteksi selain HIST yang diusulkan tersebut menggunakan *software metrics* sebagai fitur untuk melakukan analisis *source code*, evaluasi dan deteksi pemrograman berorientasi obyek. Literatur [13], memberikan rangkuman dan penjelasan tentang beberapa *software metrics* beserta literatur yang menjadi referensi pengusulan *software metrics* tersebut. *Software metrics* yang populer digunakan untuk melakukan evaluasi pada pemrograman berorientasi obyek dibahas secara lebih mendetail pada literatur [18] dan

beberapa *software metrics* klasik dan populer seperti LOC, HCM, dan CCM (*Cyclomatic Complexity*) dibahas pada literatur [19]. Untuk penjelasan tentang *Cyclomatic Complexity* sebagai *software metrics* yang digunakan untuk mengukur kompleksitas operasi, dibahas secara lebih detail dalam literatur [20] dan [19]. Kemudian untuk mengukur kohesi suatu *method* digunakan beberapa *software metrics* seperti TCC (*Tight Class Cohession*) dan LCC (*Loose Class Cohession*) yang diperkenalkan dalam literatur [15]. Beberapa *software metrics* lain secara umum telah disebutkan pada literatur [14].

Literatur [16] menggunakan *Neural Network* untuk mengidentifikasi lokasi *bad smell*. Pada penelitian tersebut digunakan 8 *software metrics* seperti LOC, WMC, RFC, LCOM1, LCOM2, LCOM5, CBO dan DIT. Untuk menyesuaikan dengan jumlah *software metrics* yang dimasukan, maka dibuat *Neural Network* dengan 8 *neuron* sebagai layer masukan, 3 neuron sebagai *hidden layer*, dan 1 *output layer*. Pada penelitian tersebut, beberapa *bad smell* dan evolusinya seperti *Antisingleton*, *Blob*, *Class Data Should Be Private*, *Complex class*, *Spaghetti Code*, *Swiss Army Knife*, *Lazy Class*, *Long Method*, *Long Parameter List*, *Message Chains*, *Refused Parent Request*, dan *Large class* dijadikan sebagai target deteksi.

Literatur [5] menggunakan *Decision Tree* sebagai pendekatan untuk melakukan klasifikasi dan membangun *knowledge base*. Fase *training* dan deteksi dilakukan secara independen. Model *knowledge base* dibangun pada fase training. Model tersebut merepresentasikan hasil akumulasi pengetahuan yang sebelumnya dipelajari berdasarkan masukan dari pengguna. Penelitian tersebut juga menggunakan *software metrics* sebagai fitur untuk mengidentifikasi *bad smell*. Beberapa contoh *bad smell* yang ditargetkan untuk dideteksi pada penelitian tersebut adalah *long method* dan *big class*.

Beberapa jenis *bad smell* seperti *Divergent Change*, *Shotgun Surgery*, dan *Parallel Inheritance* merupakan gejala yang secara intrinsik dapat dideteksi melalui *history* dari proyek. *Bad smell* dengan tipe *Blob* and *Feature Envy* ini dapat dideteksi dengan melakukan analisis *source code* menggunakan *software metrics*. Namun, ada kemungkinan juga bahwa *Blob* dan *Feature Envy* ini dapat dideteksi dengan menggunakan *history* perubahan *source code*. Berdasarkan kemungkinan tersebut, literatur [17] menggunakan pendekatan HIST untuk mendeteksi 5 jenis *bad smell* tersebut. Dalam literatur tersebut diklaim bahwa pendekatan tersebut merupakan pendekatan pertama yang menggunakan model informasi *history* perubahan berdasarkan versi sistem. Analisis *history* yang dicontohkan dalam hal ini semisal proyek yang disimpan di CVS, SVN, dan git. Setiap terjadi perubahan *source code*, detail perubahan akan disimpan sehingga dapat dimanfaatkan untuk melakukan deteksi dengan menggunakan pendekatan HIST.



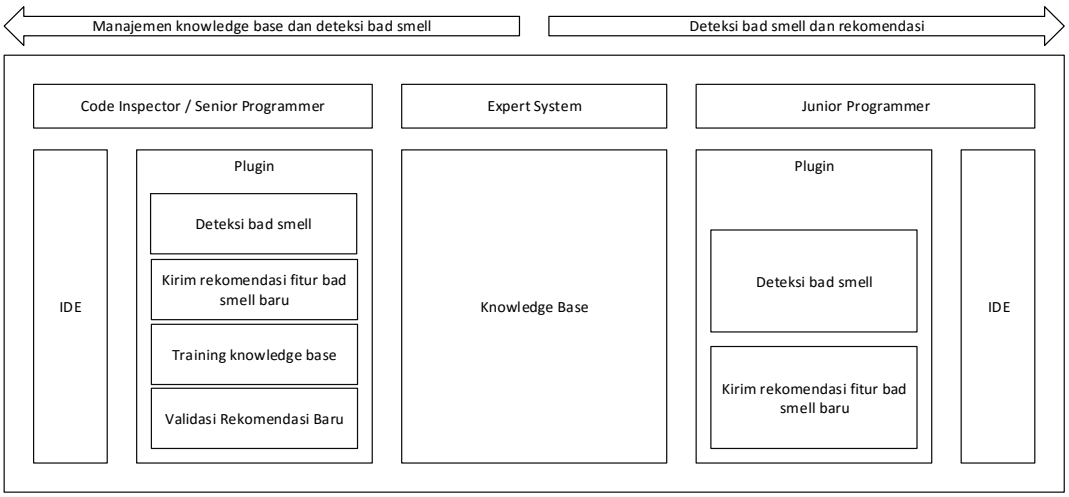
Gambar 1. Contoh *decision tree [21]*

Terinspirasi oleh penggunaan *Expert System* pada aplikasi konsultasi berbasis SMS *(Short Message Service)* pada literatur [21] dan penggunaan *decision tree* sebagai pendekatan untuk melakukan deteksi *bad smell* pada literatur [5], maka dalam kasus ini *Expert System* akan diterapkan untuk membangun *knowledge base* dari *framework* yang akan diusulkan. *Expert System* digunakan dalam kasus ini karena mampu menjawab tujuan *framework* seperti yang telah dijelaskan pada bagian pembahasan pertama. Tujuan tersebut antara lain *reasonable* dan *explainable*, *traceable*, serta *adaptive*. *Expert System* memenuhi syarat *reasonable* dan *explainable* karena menggunakan model *decision tree*, sehingga ketika sebuah *bad smell* terdeteksi dapat dijelaskan alasannya berdasarkan rute pengambilan keputusan mulai dari *root node* hingga *leaf node*. HIST juga memenuhi syarat tersebut karena dapat menjelaskan hasil deteksi berdasarkan kronologi yang diperoleh dari informasi *history* perubahan *source code*. Sementara informasi yang serupa tidak dapat dijelaskan dengan metode *Neural Network*. Untuk setiap metode yang dibahas sebelumnya termasuk *Expert System* pasti memenuhi syarat *traceable* karena setiap metode tersebut bertujuan untuk mencari lokasi *bad smell* dalam *source code*. Kemudian untuk syarat

adaptive, semua metode yang telah dibahas sebelumnya memenuhi syarat tersebut. Setiap algoritma *machine learning* seperti *Neural Network* dan *Expert System* memenuhi syarat *adaptive* karena model *knowledge base* yang dihasilkan bersifat akumulatif berdasarkan pengalaman pengguna. Sementara pendekatan HIST tergolong sebagai metode yang adaptif juga karena memanfaatkan informasi *history* perubahan *source code* untuk melakukan analisa *heuristic*. Dalam pembahasan makalah ini, desain *framework* difokuskan terhadap deteksi beberapa *bad smell* yang dapat ditangani dengan beberapa *software metrics*. Meskipun memenuhi semua tujuan yang diajukan, HIST hanya dapat mendeteksi keberadaan *bad smell* yang berkaitan erat dengan *history* perubahan *source code*. Oleh karena itu dalam kasus ini *Expert System* lebih tepat untuk digunakan.

4. DESAIN FRAMEWORK DETEKSI BAD SMELL

Berdasarkan permasalahan yang telah disampaikan pada bagian pertama, untuk menghadapi variasi perbedaan persepsi setiap *programmer* dibutuhkan sebuah *framework* pemrograman yang dapat digunakan dalam tim sebagai mediator. Tujuan dari dibentuknya *framework* tersebut telah dijelaskan pada bagian pendahuluan. Secara umum, *framework* yang diusulkan di dalam makalah ini dapat dilihat pada gambar 2. Pelaku atau pengguna *framework* diklasifikasikan menjadi dua bagian. Pengguna pertama bertindak sebagai *code inspector* ataupun *senior programmer*. Sementara pengguna kedua bertindak sebagai *junior programmer*. Jembatan yang menghubungkan pengguna pertama dan pengguna kedua adalah *knowledge base* yang mampu bertindak sebagai *Expert System*. Masing-masing pengguna *framework* akan saling berinteraksi menggunakan *Integrated Development Environment* (IDE) untuk melakukan pengembangan dan pengubahan *source code*. Untuk mengakses *knowledge base*, maka pengguna akan dibantu dengan oleh *plugin* yang telah terintegrasi dengan IDE. *Plugin* tersebut berperan untuk memainkan fungsi deteksi dan manajemen *bad smell*. *Plugin* akan memberikan fungsionalitas manajemen *bad smell* yang berbeda berdasarkan peran pengguna dalam tim.



Gambar 2. *Framework* deteksi *bad smell* dalam pemrograman tim

Pengguna pertama diasumsikan sebagai seseorang yang dipercaya memiliki kemampuan dan pengalaman lebih dalam pemrograman. Dalam hal ini yang disebut pengguna pertama adalah *code-inspector* atau *senior programmer*. Pengguna pertama memiliki kendali penuh atas pengelolaan *knowledge base* dari *framework*. Beberapa tugas yang dilakukan oleh pengguna pertama adalah deteksi *bad smell*, pengiriman rekomendasi fitur *bad smell* baru, *training knowledge base*, dan penyetujuan rekomendasi apabila terdapat fitur *bad smell* baru yang direkomendasi oleh pengguna lain. Tentu saja pengguna pertama memiliki hak untuk menolak ataupun menerima rekomendasi tersebut. Kemudian pengguna kedua diasumsikan sebagai seseorang yang memiliki lebih sedikit pengalaman dan kemampuan pemrograman jika dibandingkan dengan pengguna pertama. Pengguna kedua hanya dapat mengakses beberapa fungsi seperti deteksi *bad smell* dan pengiriman rekomendasi fitur *bad smell* baru jika suatu saat pengguna kedua tersebut menemukan model *bad smell* yang menurut pengguna kedua seharusnya terklasifikasikan sebagai *bad smell* namun tidak terdeteksi oleh *framework*. Kemudian fitur yang dikirimkan pengguna kedua akan dimasukan ke dalam *knowledge base* jika pengguna pertama menyetujui usulan pengguna kedua tersebut.

Berdasarkan skenario *framework* yang telah disampaikan, maka beberapa blok komponen dan fungsi utama yang harus ada di dalam *framework* ini adalah deteksi *bad smell*, *training knowledge base*, kirim rekomendasi fitur baru, fungsi validasi rekomendasi baru, dan yang paling penting adalah *knowledge base*. Secara lebih detail, masing-masing blok tersebut akan dibahas dalam sub-bagian selanjutnya.

4.1. *Knowledge Base*

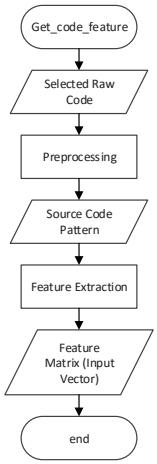
Expert System digunakan sebagai metode untuk membangun *knowledge base*. Model representasi *knowledge base* yang mewakili *Expert System* adalah *decision tree*. Seperti yang ditunjukan pada gambar 1, struktur *decision tree* terdiri atas beberapa *node* yang tersusun dari *root node* yang merupakan *decision node* pertama hingga *leaf node* yang merepresentasikan keputusan. Dalam kasus deteksi *bad smell*, *software metrics* direpresentasikan sebagai *decision node*. Keputusan akhir bahwa suatu bagian dari *source code* adalah *bad smell* atau bukan direpresentasikan sebagai *leaf node*. Di antara *decision node* dengan *decision node* atau *decision node* dengan *leaf node* terdapat *branch* yang berfungsi untuk menghubungkan antar *node* dan memberikan pilihan terhadap vektor masukan. Dalam hal ini yang dimaksud vektor masukan adalah sekumpulan nilai *software metrics* yang merepresentasikan *source code* yang sedang diperiksa. Setiap *branch* memiliki parameter standar yang berfungsi untuk mengklasifikasikan nilai masukan. Parameter standar ini disebut sebagai *treshold*.

Dalam pembahasan sebelumnya, beberapa *bad smell* akan dideteksi dengan memanfaatkan *software metrics*. Beberapa kasus *bad smell* yang akan dimodelkan adalah *Large Class*, *Brain Class*, *God Class*, *Lazy Class*, *Long Method*, *God Method*, dan *Brain Method*. Masing-masing *bad smell* tersebut memiliki ciri-ciri khusus. Berdasarkan ciri-ciri yang telah disebutkan pada pembahasan sebelumnya, maka beberapa *software metrics* yang bersesuaian dengan ciri-ciri tersebut dapat digunakan untuk membangun model *knowledge base* pada masing-masing *bad smell* yang akan dideteksi.

- *Long Method*
  Dalam kesimpulan yang diberikan oleh literatur [6], pada salah satu poin disebutkan bahwa LOC merupakan *software metric* yang sangat penting, namun seharusnya hal tersebut bukan satu-satunya kriteria untuk untuk mengidentifikasi *Long Method*. Meskipun demikian, dalam kasus ini LOC akan digunakan sebagai satu-satunya *decision node*. Hal tersebut dikarenakan LOC merupakan fitur utama untuk mengidentifikasi *Long Method*. Sementara itu, fitur-fitur lain seperti kompleksitas operasi sudah termasuk sebagai ciri-ciri dari *Brain Method*. Selain itu literatur tersebut tidak mengharuskan untuk menggunakan *software metrics* lain. Literatur tersebut hanya mengatakan seharusnya terdapat kriteria lain untuk mengidentifikasi lain. Hal tersebut berarti bahwa tidak wajib untuk menggunakan *software metrics* yang lain untuk mengidentifikasi *Long Method*.
- *Brain Method*
  Literatur [7] melakukan pengukuran menggunakan beberapa *software metrics* yang berkaitan dengan kompleksitas operasi dari *Brain Method* seperti LOC untuk menyatakan jumlah baris *source code*, CYCLO untuk menyatakan *cyclomatic complexity* atau jumlah *path* linier independen dalam sebuah *method*, MAXNESTING untuk menyatakan jumlah level *nesting* dari struktur kontrol dalam operasi, dan NOAV untuk menyatakan jumlah variabel yang diakses. Dengan demikian, dalam kasus ini *software metrics* tersebut juga dapat dijadikan sebagai *decision node* untuk membangun model *knowledge base* karena *software metrics* tersebut memiliki pengaruh terhadap *Brain Method*.
- *God Method*
  Literatur [6] menjelaskan bahwa *God Method* memiliki kecenderungan terhadap kompleksitas operasi. Maka beberapa *software metrics* yang dapat dimanfaatkan untuk mengidentifikasi *God Method* hampir sama seperti *Brain Method*. Namun *God Method* cenderung mengutamakan masalah kompleksitas operasi tanpa mempedulikan jumlah *statement* dalam *method*, sehingga tidak memerlukan LOC. Diantara beberapa *software metrics* yang dapat dipakai untuk mengidentifikasi *God Method* adalah seperti CYCLO untuk menyatakan *cyclomatic complexity* atau jumlah *path* linier independen dalam sebuah *method*, MAXNESTING untuk menyatakan jumlah level percabangan dari struktur kontrol dalam operasi, dan NOAV untuk menyatakan jumlah variabel yang diakses.
- *God Class* dan *Brain Class*
  Dalam kajian yang disajikan pada literatur [7] disebutkan bahwa *God Class* dan *Brain Class* hanya memiliki perbedaan dalam hal jumlah kohesi dan akses atribut ke variabel di luar *class*, sehingga untuk membangun model deteksi *God Class* dan *Brain Class* dibutuhkan beberapa *software metrics* yang sama. Dalam literatur [7] disebutkan beberapa *software metrics* seperti WMC untuk menyatakan jumlah kompleksitas operasi dari semua *method* yang ada dalam sebuah *class*, TCC untuk menyatakan jumlah relatif dari *method* yang secara langsung dalam sebuah *class*, ATFD untuk menyatakan jumlah atribut *class* lain baik yang diakses secara langsung ataupun melalui *accessor method* dalam sebuah *class* tertentu, NBM untuk menyatakan jumlah *Brain Method* dalam sebuah *class*, dan LOC untuk menyatakan jumlah baris kode.
- *Large Classs*
  Berdasarkan keterangan yang disampaikan pada literatur [7], *God Class* dan *Brain Class* merupakan bentuk turunan dari *Large Class*. Sehingga apabila sebuah *class* terdeteksi sebagai *God Class* ataupun *Brain Class* secara otomatis maka akan terdeteksi sebagai *Large Class*. Oleh karena itu untuk mendeteksi *Large Class*, dapat digunakan GC untuk merepresentasikan bahwa sebuah *class* merupakan *God Class*

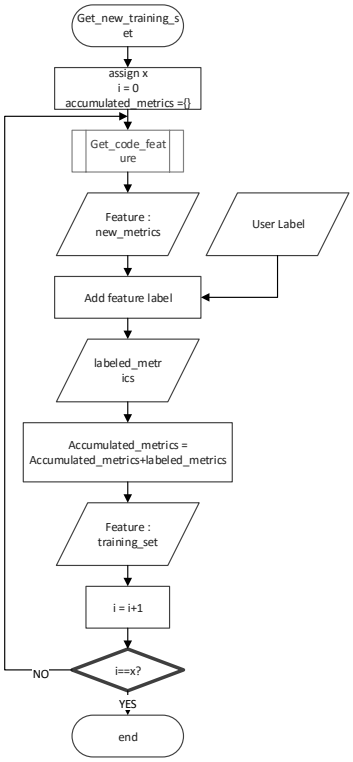
atau bukan, kemudian BC untuk merepresentasikan bahwa sebuah *class* merupakan *Brain Class* atau bukan.

- *Lazy Class*
  Dengan melihat metode yang dilakukan pada literatur [10] untuk mendeteksi *Lazy Class*, maka dapat disimpulkan bahwa beberapa *software metrics* memiliki relasi terhadap ciri-ciri *Lazy Class*. Beberapa *software metrics* tersebut misalnya adalah NOM untuk merepresentasikan jumlah *method* yang ada dalam sebuah *class*, LOC untuk merepresentasikan jumlah baris kode dalam sebuah *class*, WMC untuk merepresentasikan jumlah kompleksitas operasi dari semua *method* yang ada dalam sebuah *class*, DIT yang mengukur kedalaman hieraki *inheritance* dari *class* tesebut berdasarkan jumlah *class* induknya, dan CBO yang berfungsi mengukur relasi *class* yang akan diukur dengan *class* lain (*coupling*). Dengan demikian komposisi *software metrics* tersebut dapat digunakan untuk membangun *knowledge base* yang mampu mengenali *Lazy Class*.



Gambar 3. Mendapatkan fitur dari *source code*

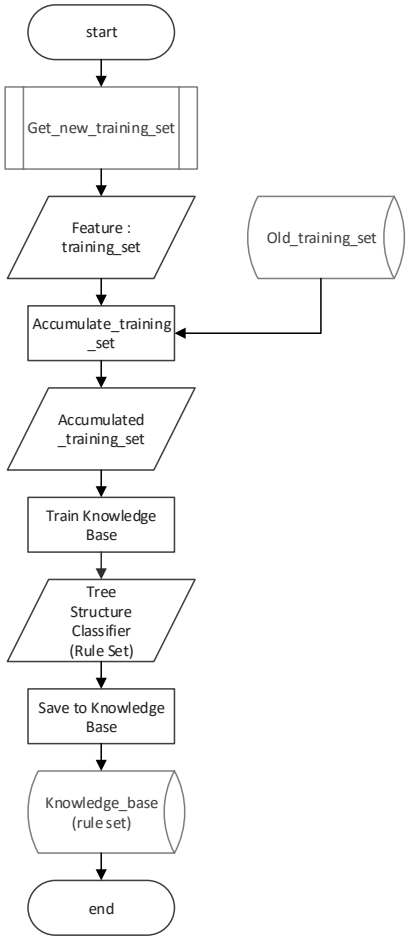
Sesuai dengan kriteria pada masing-masing *bad smell*, maka setiap *software metrics* yang saling berkaitan dengan *bad smell* tersebut akan digunakan untuk membuat struktur *decision tree*. Pada akhirnya, setiap *bad smell* mempunyai *decision tree* masing-masing yang selanjutnya akan dimanfaatkan oleh setiap pengguna *framework* sebagai referensi pengenalan *bad smell* melalui *plugin* masing-masing pengguna. Proses pembangkitan *decision tree* tersebut terjadi melalui proses *training knowledge base*.



Gambar 4. Menginisiasi *training set* baru

4.2. *Training Knowledge Base*

Training knowledge base bertujuan untuk menanamkan *knowledge base* pada *framework* yang akan dibangun. Dengan tersedianya *knowledge base* tersebut, *framework* mampu melakukan deteksi *bad smell* dengan memanfaatkan *knowledge base* yang telah dibangun tersebut sebagai acuan deteksi. Proses mulai dari persiapan untuk melakukan *training* hingga terbentuknya *decision tree* dapat digambarkan melalui *flowchart* seperti pada gambar 5. Untuk setiap *bad smell* memerlukan *x* data *training* berupa vektor $v_{bad\ smell}$ yang terdiri dari beberapa *software metrics* terkait dengan *bad smell* yang ingin dikenali sejumlah *x* buah. Untuk *Long Method* maka diperlukan $v_{long\_method} = \langle LOC, DECISION \rangle$ sejumlah *x* yang di antara *x* vektor tersebut terdiri dari beberapa sampel positif dan beberapa sampel negatif. Untuk *Brain Method* maka diperlukan sejumlah *x* vektor $v_{brain\_method} = \langle LOC, CYCLO, MAXNESTING, NOAV, DECISION \rangle$ yang di antara *x* vektor tersebut terdiri dari beberapa sampel positif dan beberapa sampel negatif. Untuk *God Method* maka diperlukan $v_{god\_method} = \langle CYCLO, MAXNESTING, NOAV, DECISION \rangle$ sejumlah *x* yang di antara *x* vektor tersebut terdiri dari beberapa sampel positif dan beberapa sampel negatif. Untuk *God Class* dan *Brain Class* maka diperlukan $v_{god\_class\ ||brain\_class} = \langle WMC, TCC, ATFD, NBM, LOC, DECISION \rangle$ sejumlah *x* yang di antara *x* vektor tersebut terdiri dari beberapa sampel positif dan beberapa sampel negatif. Untuk *Large Class* maka diperlukan $v_{large\_class} = \langle GC, BC, DECISION \rangle$ sejumlah *x* yang di antara *x vector* tersebut terdiri dari beberapa sampel positif dan beberapa sampel negatif. Dan untuk *Lazy Class*, maka diperlukan $v_{lazy\_class} = \langle NOM, LOC, WMC, DIT, CBO, DECISION \rangle$ sejumlah *x* yang di antara *x* vektor tersebut terdiri dari beberapa sampel positif dan beberapa sampel negatif juga.

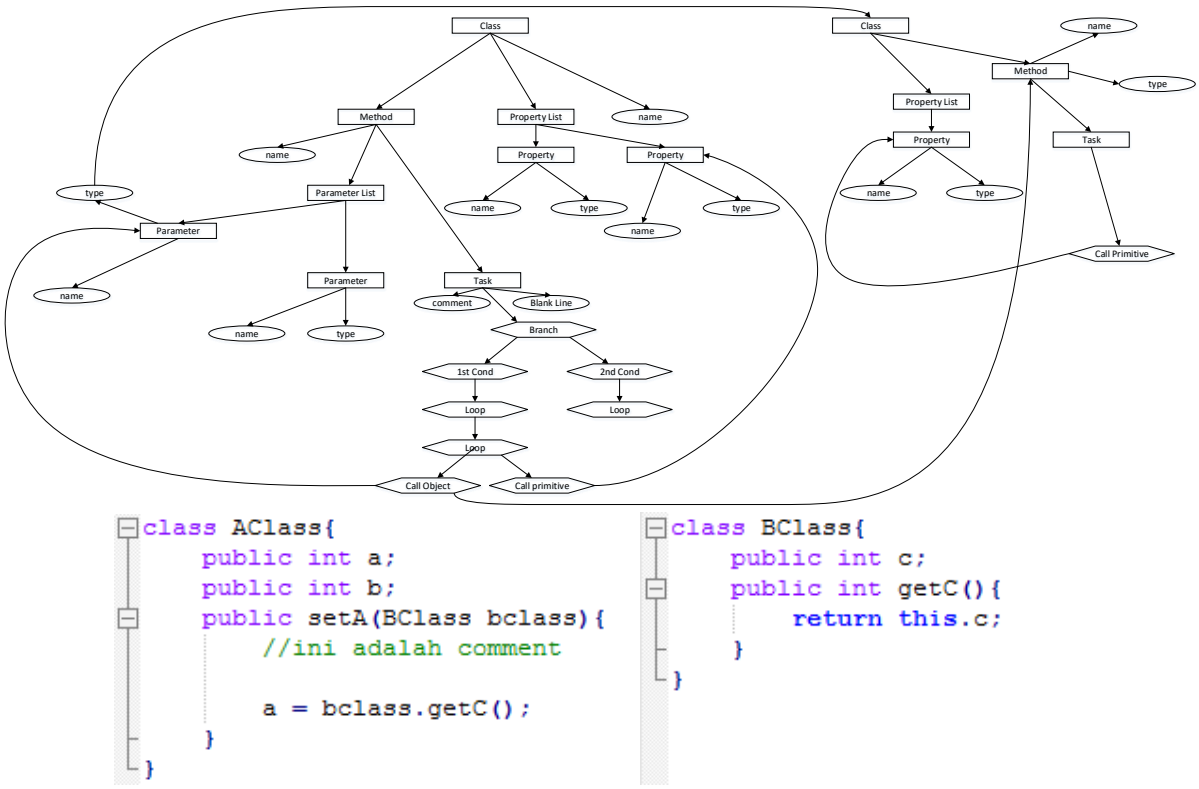


Gambar 5. Proses *training knowledge base* sistem pakar

Dalam *training knowledge base* ini terdapat tiga proses penting yang berjalan secara sekuensial. Proses tersebut secara berturut-turut yaitu *preprocessing*, ekstraksi fitur, dan *data training*. Masukan yang diumpankan kepada *framework* pertama kali adalah *source code* mentah. Kemudian dari *source code* mentah mengalami tiga proses yang telah disebut secara berturut turut. Hasilnya adalah struktur *knowledge base* yang berupa *decision tree*. Proses *preprocessing* dan ekstraksi fitur pada *flowchart* gambar 3, merupakan bagian dari proses untuk mendapatkan *training set* baru yang diilustrasikan dengan *flowchart* pada gambar 4. Proses data *training* diilustrasikan melalui gambar 5.

Proses pertama adalah *preprocessing*. Dalam proses *preprocessing* ini *source code* digunakan sebagai masukan. Apabila diperlukan *training* untuk mendeteksi *bad smell* pada level *method*, maka dalam hal ini akan dipilih *source code* pada baris tertentu yang berupa *method* sebagai masukan. Sedangkan apabila diperlukan *training* untuk mendeteksi *class*, maka *source code* yang dipilih sebagai masukan adalah *source code* pada baris tertentu yang merepresentasikan sebuah *class*. Pada proses *preprocessing*, setiap bagian dari *source code* masukan akan diubah dan diuraikan menjadi sebuah *tree graph* terstruktur yang menyatakan suatu hierarki dan fungsi keanggotaan. Masing-masing keanggotaan dinyatakan sebagai entitas yang memiliki atribut. Proses *preprocessing* dapat diilustrasikan seperti pada gambar 6. Hasil *preprocessing* dari *source code* yang berbentuk *tree graph* itu disebut sebagai *source code pattern* (pola *source code*). Dengan bentuk *source code pattern* ini, data akan lebih mudah untuk diproses ke tahap berikutnya.

Tahap selanjutnya adalah proses ekstraksi fitur. Dalam proses ekstraksi fitur ini *source code pattern* digunakan sebagai masukan. Proses ekstraksi fitur bertujuan mendapatkan fitur *source code* yang berupa *software metrics*. *Source code pattern* kemudian akan diolah menjadi beberapa *software metrics* tertentu sesuai dengan kebutuhan deteksi masing-masing *bad smell*. Misalnya untuk membangun model *Lazy Class*, maka *software metrics* yang diperlukan untuk proses pada tahap selanjutnya adalah NOM, LOC, WMC, DIT, dan CBO. Dengan demikian, untuk model *Lazy Class* proses ekstraksi fitur akan menghasilkan *vector* $v_{lazy\_class} = \langle NOM, LOC, WMC, DIT, CBO, DECISION \rangle$.




```
class AClass{
    public int a;
    public int b;
    public setA(BClass bclass){
        //ini adalah comment

        a = bclass.getC();
    }
}
```

```
class BClass{
    public int c;
    public int getC(){
        return this.c;
    }
}
```

Gambar 6. Ilustrasi *preprocessing* dari *source code* menjadi *tree graph* (*source code pattern*).

Proses *preprocessing* dan ekstraksi fitur dilakukan secara berulang ulang dengan masukan *source code* yang berbeda. Proses tersebut diulang sebanyak x kali sesuai dengan *request* dari pengguna. Pada setiap perulangan, fitur *vector* yang didapatkan terus diakumulasikan diberikan label oleh pengguna berupa DECISION, sehingga terbentuk *training set* baru seperti pada gambar 4. Bentuk dari DECISION adalah kategori klasifikasi yang bermacam-macam tergantung pada kasus. Contohnya DECISION yang mungkin untuk mendeteksi *Lazy Class* terdapat dua kemungkinan yang merepresentasikan *leaf node* yaitu $\{YA, TIDAK\}$. Namun DECISION untuk melakukan deteksi pada *God Class* dan *Brain Class* kemungkinan dapat direpresentasikan dengan tiga kategori seperti $\{GOD\ CLASS, BRAIN\ CLASS, TIDAK\}$. Selanjutnya *training set* baru diakumulasikan dengan *training* set lama yang sebelumnya telah disimpan pada *database*.

$$v_{new\_training\_set} = \{v_{bad\_smell\_1}, v_{bad\_smell\_2}, v_{bad\_smell\_3}, \cdots, v_{bad\_smell\_x}\}$$
$$v_{training\_set} = v_{new\_training\_set} + v_{old\_training\_set}$$

Proses yang ketiga adalah *data training*. *Training set* baru yang telah didapatkan akan dijadikan sebagai masukan. Proses *data training* ini secara umum berfungsi untuk membangun *knowledge base* berupa *decision tree* dari *training set* baru tersebut. Sebelum melakukan *data training*, maka pengguna terlebih dahulu mendefinisikan *threshold* statis untuk melakukan *training*. Masing-masing jenis *software metrics* yang menjadi fitur dalam setiap elemen $v_{training\_set}$ akan diklasifikan menurut *threshold* masing-masing sesuai dengan kriteria *bad smell*. Untuk melakukan *training* pada *bad smell Large Class* maka untuk setiap vektor *Large Class* ke-i $v_{training\_set}[i] = \langle GC, BC, DECISION \rangle$ nilai GC dan BC dalam *training* set ke-i tersebut akan diubah menjadi beberapa kategori seperti TRUE dan FALSE tergantung pada *vector threshold* yang didefinisikan oleh pengguna. Sedangkan untuk klasifikasi kategori DECISION diputuskan oleh pengguna secara pribadi. Seperti yang telah disebutkan sebelumnya, kategori DECISION bergantung pada jumlah keputusan yang mungkin terjadi dalam deteksi jenis *bad smell* tertentu.

Berikut merupakan contoh kasusnya. Pengguna *framework* mendefinisikan *threshold* untuk *data training* pada *bad smell Large Class*, sehingga masing-masing nilai GC dan BC secara berturut-turut pada vektor *threshold* didefinisikan sebagai $v_{threshold} = \langle GC \geq 1, BC \geq 1, DECISION \rangle$. Selanjutnya apabila terdapat $v_{training\_set}[i] = \langle 1,0, DECISION \rangle$ sebagai sampel untuk melakukan *data training*, maka secara berturut-turut nilai GC dan BC dalam vektor tersebut akan diubah dalam bentuk vektor kategori sehingga hasil akhirnya adalah $v_{training\_set}[i] = \langle TRUE, FALSE, DECISION \rangle$. GC menjadi TRUE karena nilai input lebih besar atau sama dengan dibandingkan dengan *threshold* untuk GC, sedangkan BC menjadi FALSE karena nilai BC lebih kecil dari nilai *threshold* untuk BC yang didefinisikan pengguna. Apabila pengguna menginisiasi nilai YA untuk DECISION pada $v_{training\_set}[i]$ tersebut, maka bentuk akhir *vector* ke-i tersebut adalah $v_{training\_set}[i] = \langle TRUE, FALSE, YA \rangle$. Proses tersebut akan terus diulang sejumlah x data masukan sehingga dihasilkan sejumlah x *tupple* data seperti pada contoh tabel 1. Pada tabel 1 dicontohkan nilai $x = 3$. Setelah itu data untuk proses *training* siap untuk diolah.

Tabel 1. Contoh data *training* untuk *Lazy Class* sejumlah x *tupple*

$x = 3$

i	GC	BC	DECISION
1	TRUE	TRUE	YA
2	TRUE	FALSE	YA
3	FALSE	FALSE	TIDAK

Kumpulan data seperti pada tabel 1 tersebut selanjutnya akan diolah hingga terbentuk *knowledge base* dari *Expert System* yang dimodelkan dengan bentuk *decision tree*. Cara pengolahan data tersebut adalah dengan mencari nilai entropi dan gain dari setiap variabel *software metrics* yang mewakili. Teknik Gain dan entropi ini berfungsi untuk mencari variabel yang paling representatif dan relevan terhadap *leaf node* (keputusan). Entropi menunjukkan kemurnian informasi sebuah variabel terhadap keputusan. Dengan demikian, maka variabel yang memiliki nilai gain paling tinggi (relevan) dapat dimanfaatkan sebagai *root node*. Dengan teknik entropi dan gain, *decision tree* yang dihasilkan akan lebih efisien dan lebih ringkas. Dalam contoh kasus pada tabel 1, maka variabel yang harus dicari entropinya adalah GC, BC, dan DECISION.

Untuk variabel acak Y dengan probabilitas $P(y)$, entropi adalah jumlah rata-rata atau harapan informasi yang dapat diperoleh dengan mengamati y. Secara matematis entropi dari variabel acak Y dapat dirumuskan sebagai berikut:

$$H(Y) = -\sum_{y} P(y) \log_2 P(y)$$

Untuk entropi bersyarat dapat dirumuskan sebagai berikut:

$$H(Y|X) = -\sum_{x} P(x) \sum_{y} P(y|x) \log_2 P(y|x)$$

dimana $H(Y|X)$ merupakan entropi yang tersisa dari Y dengan mengetahui X. Dengan mengetahui $H(Y)$ dan $H(Y|X)$ maka dapat diperolah informasi mutual (*gain*) antara Y dengan X yang dapat dirumuskan secara matematis sebagai berikut:

$$I(Y;X) = H(Y) - H(Y|X)$$

$I(Y;X)$ akan bernilai 0 jika Y tidak menyediakan informasi apapun terhadap X atau dapat diartikan juga bahwa $P(x)$ dan $P(y)$ adalah independen.

Apabila dikaitkan dengan kasus *Large Class* sebelumnya, dengan memanfaatkan persamaan di atas maka perhitungan entropi dan gain dapat dilakukan sebagai berikut:

$$H(DECISION) = -P(DECISION = YA) \times \log_2 P(DECISION = YA) - P(DECISION = TIDAK) \times \log_2 P(DECISION = TIDAK)$$

$$H(DECISION) = -\frac{2}{3} \times \log_2 \frac{2}{3} - \frac{1}{3} \times \log_2 \frac{1}{3} = 0.92$$

Untuk setiap kemungkinan GC maka:

$$H(DECISION|GC = FALSE) = -\frac{1}{1} \log_2 \frac{1}{1} = 0$$

$$H(DECISION|GC = TRUE) = -\frac{2}{2} \log_2 \frac{2}{2} = 0$$

sehingga entropi DECISION jika diketahui GC adalah:

$$H(DECISION|GC) = \frac{1}{3} \times 0 + \frac{2}{3} \times 0 = 0$$

Untuk setiap kemungkinan BC maka:

$$H(DECISION|BC = FALSE) = -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = 1$$

$$H(DECISION|BC = TRUE) = -\frac{1}{1} \log_2 \frac{1}{1} = 0$$

sehingga entropi DECISION jika diketahui BC adalah:

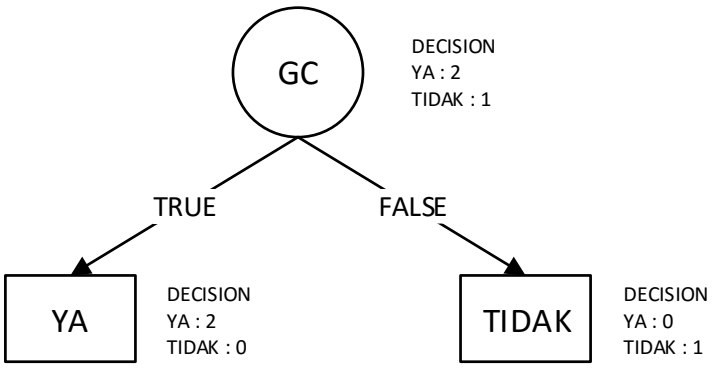
$$H(DECISION|BC) = \frac{2}{3} \times 1 + \frac{1}{3} \times 0 = 0.67$$

Untuk setiap GC dan BC, maka gain masing-masing adalah:

$$I(DECISION;GC) = H(DECISION) - H(DECISION|GC) = 0.92 - 0 = 0.92$$

$$I(DECISION;BC) = H(DECISION) - H(DECISION|BC) = 0.92 - 0.67 = 0.25$$

Dengan melihat perbandingan gain antara GC dengan BC, maka berdasarkan informasi data yang diberikan pada tabel 1 terbukti bahwa GC memiliki nilai gain yang lebih besar apabila dibandingkan dengan BC. Dengan demikian maka GC akan dijadikan sebagai *root node* (*node* level 0) sedangkan BC akan dijadikan sebagai *decision node* level 1 jika diperlukan. Namun dengan contoh kasus di atas, GC sudah cukup representatif untuk memberikan DECISION sehingga BC tidak perlu dilibatkan dalam pengambilan keputusan. Dengan demikian, maka *decision tree* yang dibangun akan terlihat seperti pada gambar 7.



Gambar 7. *Decision Tree* berdasarkan contoh pada tabel 1

Jumlah dan variasi data untuk *training* akan sangat berpengaruh terhadap *decision tree* yang dihasilkan. Dengan demikian semakin banyak dan akurat data untuk *training*, maka proses *training knowledge base* akan menghasilkan *decision tree* yang semakin representatif juga. Variasi dari data untuk *training* akan sangat mempengaruhi nilai entropi dan gain. Misal data *training* pada tabel 1 ditambahkan dengan sebuah *vector training set* $v_{training\_set}[i] = \langle FALSE, TRUE, YA \rangle$, maka entropi, gain, dan *decision tree* yang dihasilkan mungkin akan berbeda. Dengan cara kerja yang sama seperti yang telah dicontohkan pada studi kasus *Large Class* di atas, maka untuk membangun *knowledge base* beberapa *bad smell* lain yang telah ditargetkan pada pembahasan sebelumnya perlu memperhatikan beberapa hal berikut:

- *Long Method*
  Untuk *Long Method*, maka masukan dari proses *training knowledge base* adalah *source code pattern* dari *method* tertentu. Hal yang perlu dipersiapkan adalah sejumlah x vektor *training set* dimana setiap *training set* ke-i merupakan *tupple* $v_{training\_set}[i] = \langle LOC, DECISION \rangle$. Vektor *treshold* untuk LOC juga perlu untuk diinisiasi terlebih dahulu sebelum melakukan *data training*. Kategori LOC disesuaikan dengan kebutuhan pengguna. Contohnya LOW, NORMAL, dan HIGH ataupun NORMAL dan HIGH saja. Untuk DECISION dalam kasus *knowledge base* untuk *Long Method* ini hanya memerlukan dua keluaran kategori saja yaitu YA dan TIDAK.
- *Brain Method*
  Untuk *Brain Method*, maka masukan dari proses *training knowledge base* adalah *source code pattern* dari *method* tertentu. Sebelum melakukan *data training*, beberapa x vektor *training set* dimana $v_{training\_set}[i] = \langle LOC, CYCLO, MAXNESTING, NOAV, DECISION \rangle$ perlu dipersiapkan. Untuk setiap elemen *tupple* dari *training set* perlu untuk dikategorikan terlebih dahulu berdasarkan vektor *threshold* sesuai dengan kebutuhan pengguna. Dalam kasus ini, DECISION hanya memerlukan dua kategori keluaran saja yaitu YA dan TIDAK.
- *God Method*
  Untuk *God Method*, maka masukan dari proses *training knowledge base* adalah *source code pattern* dari *method* tertentu. Sejumlah x vektor *training set* perlu dipersiapkan dimana setiap vektor *training set* tersebut merupakan *tupple* $v_{training\_set}[i] = \langle CYCLO, MAXNESTING, NOAV, DECISION \rangle$. Untuk setiap elemen *tupple* dari *training set* terlebih dahulu dikategorikan berdasarkan vektor *threshold* sesuai dengan kebutuhan pengguna. Dalam kasus ini hanya diperlukan dua kategori keluaran untuk DECISION, yaitu YA dan TIDAK.
- *God Class* dan *Brain Class*
  God Class dan *Brain Class* merupakan *bad smell* pada level *class*, sehingga masukan dari proses *training knowledge base* adalah *source code pattern* dari *class* tertentu. Sejumlah x vektor *training set* yang tersusun dalam *tupple* $v_{training\_set}[i] = \langle WMC, TCC, ATFD, NBM, LOC, DECISION \rangle$ perlu untuk dipersiapkan. Untuk setiap elemen *tupple* dari *training set* terlebih dahulu dikategorikan berdasarkan vektor *threshold* sesuai dengan kebutuhan pengguna. Dalam kasus ini, diperlukan tiga kategori keluaran yang dapat merepresentasikan DECISION yaitu GC untuk merepresentasikan apakah sebuah *class* masukan merupakan *God Class*, BC untuk merepresentasikan apakah sebuah *class* masukan merupakan *Brain Class*, dan TIDAK untuk merepresentasikan *class* masukan yang tidak termasuk sebagai GC dan BC.
- *Large Classs*
  Large Class merupakan *bad smell* pada level *class*, sehingga masukan dari proses *training knowledge base* yang dibutuhkan adalah *source code pattern* dari sebuah *class* tertentu. Sejumlah x vektor *training set* diperlukan dalam bentuk *tupple* dimana setiap training set ke-i dituliskan sebagai $v_{training\_set}[i] = \langle GC, BC, DECISION \rangle$. GC dan BC masing-masing memiliki dua kategori yaitu TRUE dan FALSE. Untuk merepresentasikan hasil, maka dalam kasus ini DECISION juga memerlukan dua kategori keluaran yaitu YA dan TIDAK.
- *Lazy Class*
  Lazy Class merupakan salah satu *bad smell* pada level *class*. Maka masukan dari proses *training knowledge base* yang dibutuhkan adalah *source code pattern* dari *class* tertentu. Untuk melakukan *training knowledge base* diperlukan sejumlah x *training set* dalam bentuk *tupple* dimana setiap *tupple* ke-i dapat dituliskan sebagai $v_{training\_set}[i] = \langle NOM, LOC, DIT, WMC, CBO, DECISION \rangle$. Pengguna terlebih dahulu harus menentukan *threshold* untuk memberikan kategori setiap *tupple* dari *training set*. Dalam kasus ini hanya diperlukan dua kategori keluaran DECISION yaitu YA dan TIDAK.

Apabila sebuah *bad smell* dapat dianalisis secara leksikal maupun struktural, maka akan lebih mudah untuk mendapatkan fitur dari *source code* berupa *software metrics*. Dengan demikian, maka *knowledge base* dari beberapa *bad smell* yang lain juga dapat dimodelkan dengan teknik yang sama seperti yang telah dicontohkan selama *bad smell* tersebut dapat direpresentasikan dalam bentuk *software metrics*.

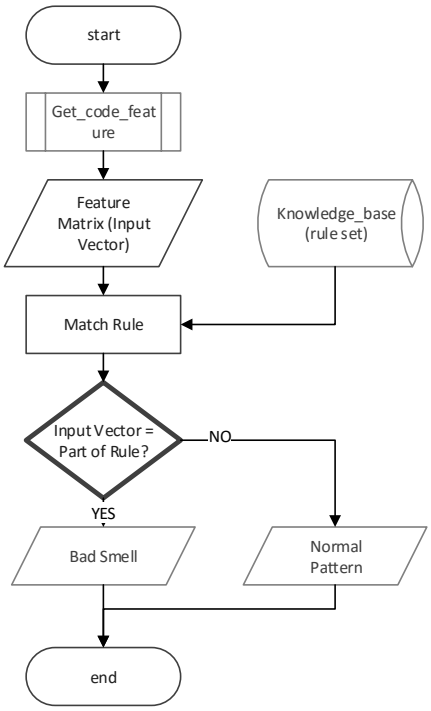
4.3. Deteksi *Bad Smell*

Deteksi *bad smell* dapat dilakukan apabila setiap *bad smell* yang ingin dideteksi telah dimodelkan melalui proses *training knowledge base*. Struktur *decision tree* yang telah dibangun dimanfaatkan untuk melakukan klasifikasi dengan masukan vektor uji. Pada dasarnya, vektor uji memiliki bentuk yang sama dengan representasi *tupple* pada *training set* yang telah dibahas sebelumnya. Proses umum untuk melakukan deteksi *bad smell* secara keseluruhan dapat direpresentasikan melalui gambar 8.

Hal pertama yang perlu dilakukan dalam proses deteksi adalah ekstraksi fitur dari *source code* yang dapat dijelaskan seperti pada gambar 3. Untuk melakukan ekstraksi fitur, maka hal yang diperlukan adalah potongan atau bagian *source code* yang merepresentasikan struktur tertentu semisal *class* ataupun *method*. Keluaran yang

dihasilkan adalah sebuah *tupple* berupa vektor uji dalam bentuk vektor *Large Class*. Misal, untuk mendeteksi *Large Class*, maka masukan dari proses ekstraksi fitur ini adalah *source code* sebuah *class*. Sedangkan, keluaran yang dihasilkan adalah vektor uji dari *class* tersebut dalam bentuk vektor *Large Class* seperti $v_{test\_large\_class}[i] = \langle GC, BC, DECISION \rangle$. Perbedaan dari vektor uji dengan *training set* adalah inisiasi dari DECISION. Di dalam vektor uji, DECISION diinisiasi setelah proses deteksi selesai. Namun pada *training set*, inisiasi DECISION dilakukan oleh pengguna sebelum atau ketika melakukan proses *data training*.

Setelah vektor uji didapatkan, maka vektor uji tersebut akan dibandingkan variabel yang saling bersesuaian dengan aturan *decision tree* yang telah terbentuk sebagai *knowledge base* secara berturut-turut hingga didapatkan hasil klasifikasi (keputusan) dari bagian *source code* masukan tersebut.



Gambar 8. Deteksi *bad smell* dengan sistem pakar

4.4. Kirim Rekomendasi Fitur *Bad Smell* Baru

Kirim rekomendasi fitur *bad smell* baru merupakan salah satu fitur yang diajukan di dalam *framework* dengan target junior *programmer* dan senior *programmer* sebagai pengguna. Fitur ini berfungsi untuk mengakomodasi junior *programmer* atau pengguna kedua pada khususnya yang memiliki perhatian dan sikap terhadap keberadaan *bad smell* dalam *source code*. Senior *programmer* atau pengguna pertama merupakan seseorang yang memiliki pengalaman lebih. Namun ketelitian tidak kalah penting jika dibandingkan dengan pengalaman. Misalnya, ketelitian dalam pengambilan sampel data *training* untuk *bad smell*. Kemungkinan dimana pengguna kedua memiliki ketelitian yang lebih dibandingkan pengguna kedua tentu selalu ada. Dengan fitur ini maka pengguna kedua dapat berpartisipasi bersama dengan pengguna pertama untuk mengembangkan *knowledge base* dari *framework*. Caranya adalah dengan mengirimkan fitur dari bagian *source code* yang diduga sebagai *bad smell* tertentu berupa vektor *training set*. Proses ekstraksi fitur dari *source code* menggunakan skema *flowchart* seperti pada gambar 3. Hasil dari proses ekstraksi fitur yang berupa vektor tersebut kemudian dikirim, kemudian disimpan di dalam *framework* sambil menunggu validasi fitur tersebut oleh pengguna pertama.

4.5. Validasi Rekomendasi Baru

Vektor *training set* yang telah dikirim oleh pengguna kedua dan tersimpan di dalam *framework* merupakan kandidat atau calon *training set* yang belum valid kebenarannya. Untuk dapat dijadikan sebagai *training set* yang valid, maka pengguna pertama harus memberikan validasi terhadap calon *training set* yang telah tersimpan di dalam *framework* tersebut. Apabila pengguna pertama menganggap bahwa beberapa calon *training set* tersebut tidak valid, maka pengguna pertama berhak untuk menolak data tersebut. Namun, jika pengguna pertama menganggap bahwa data tersebut layak untuk dijadikan *training set*, maka pengguna pertama berhak untuk menyetujui dan memasukkan data tersebut sebagai *training set* yang valid.

Pengguna pertama dapat melakukan *training* ulang untuk *knowledge base* pada masing-masing *bad smell* jika pengguna pertama merasa hal tersebut perlu untuk dilakukan. Dengan demikian, *training set* yang baru dapat diikutkan dalam proses *data training*. Dengan adanya data baru tersebut, diharapkan bahwa proses deteksi menjadi

lebih akurat terhadap data masukan yang sebelumnya tidak dikenal. Dengan melakukan *traning* ulang diharapkan juga bahwa *knowledge base* yang dihasilkan akan lebih representatif terhadap berbagai macam masukan.

5. KESIMPULAN

Di dalam karya tulis ini, telah diajukan sebuah *framework* untuk melakukan deteksi beberapa *bad smell* seperti *Large Class*, *Big Class*, *God Class*, *Lazy Class*, *Long Method*, *Brain Method*, dan *God Method* untuk melakukan pemrograman dalam tim. Dengan *framework* yang telah diajukan, diharapkan proses deteksi *bad smell* bersifat:

- *Reasonable* dan *explainable*: yaitu kondisi dimana *framework* tersebut mampu memberikan penjelasan dan sebab-sebab yang rasional ketika *framework* tersebut mengklasifikasikan bagian dari *source code* sebagai *bad smell*. Dengan demikian, programmer pemula dapat mempelajari karakteristik dari *bad smell* berdasarkan penjelasan yang disampaikan oleh *framework*.
- *Traceable*: yaitu kondisi dimana *framework* tersebut mampu untuk mendeteksi letak *bad smell* di dalam *source code*.
- *Adaptive*: yaitu kondisi dimana sebuah *framework* mampu beradaptasi terhadap perubahan *knowledge base* berdasarkan pengetahuan dan pengalaman *programmer*. Dengan demikian *framework* yang dibangun mampu untuk belajar dengan cara mengevaluasi, mengurangi, ataupun menambahkan pengetahuan berdasarkan fitur *source code* tertentu yang dipilih untuk mengidentifikasi parameter *bad smell*. Nilai parameter tersebut akan dijadikan sebagai standar untuk mengklasifikasikan dan mengidentifikasi keberadaan *bad smell* dalam pemrograman berkelompok.

Expert System digunakan sebagai model untuk membangun *knowledge base* karena memiliki ketiga keunikan tersebut. *Software metrics* digunakan sebagai fitur untuk melakukan deteksi dan analisis *bad smell*. Prioritas urutan *software metrics* dalam perancangan struktur *Expert System* dihitung dengan menggunaan analisa entropi dan gain dari kumpulan informasi berupa *software metrics* yang digunakan. Teknik deteksi *bad smell* ini memungkinkan digunakan pada beberapa jenis *bad smell* lain yang dapat dianalisis secara leksikal maupun struktural, karena *bad smell* yang dapat dianalisis secara leksikal maupun struktural pada umumnya dapat direpresentasikan melalui bentuk *software metrics* terkait.

REFERENSI

[1] B. D. Bois, P. V. Gorp, A. Amsel, N. V. Eetvelde, H. Stenten dan S. Demeyer, "A Discussion of Refactoring in Research and Practice," University of Antwerp, 2004.

[2] E. V. Emden dan L. Moonen, "Java Quality Assurance by Detecting Code Smells," *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02),* pp. 97 - 106, 2002.

[3] M. V. Mäntylä, J. Vanhanen dan C. Lassenius, "Bad Smells - Humans as Code Critics," *20th IEEE International Conference on Software Maintenance (ICSM'04),* 2004.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke dan d. Roberts, Refactoring: Improving the Design of Existing Code, Boston: Addison-Wesley Longman Publishing Co., Inc, 1999.

[5] J. Kreimer, "Adaptive Detection of Design Flaws," *Electronic Notes in Theoretical Computer Science,* pp. 117-146, 2005.

[6] X. Zhao, X. Xuan dan S. Li, "An Empirical Study of Long Method and God Method in Industrial Projects," *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop,* pp. 109 - 114, 2015.

[7] S. M. Olbrich, D. S. Cruzes dan D. I. Sjøberg, "Are all Code Smells Harmful? A Study of God Classes and Brain Classes inthe Evolution of three Open Source Systems," *Software Maintenance (ICSM), 2010 IEEE International Conference,* pp. 1 - 10, 2010.

[8] M. Abbes, F. Khomh, Y. G. Gueheneuc dan G. Antoniol, "An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension," *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference,* pp. 181 - 190, 2011.

[9] K. Nongpong, "Feature Envy Factor: A Metric for Automatic Feature Envy Detection," *Knowledge and Smart Technology (KST), 2015 7th International Conference,* pp. 7 - 12, 2015.

[10] M. J. Munro, "Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code," *11th IEEE International Software Metrics Symposium (METRICS'05),* p. 15, 2005.

[11] M. Zhang, N. Baddoo dan P. Wernick, "Improving the Precision of Fowler's Definitions of Bad Smells," *Software Engineering Workshop, 2008. SEW '08. 32nd Annual IEEE,* pp. 161 - 166, 2008.

[12] “Identification of Refused Bequest Code Smells,” *Software Maintenance (ICSM), 2013 29th IEEE International Conference,* pp. 392 - 395, 2013.

[13] B. M. Goel dan P. K. Bhatia, “An Overview of Various Object Oriented Metrics,” *International Journal of Information Technology & Systems,* vol. 2, no. 1, pp. 18-27, 2013.

[14] F. A. Fontana, V. Ferme, M. Zanoni dan A. Yamashita, “Automatic Metric Thresholds Derivation for Code Smell Detection,” *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics,* pp. 44 - 53, 2015.

[15] J. M. Bieman dan B. K. Kang, “Cohesion and Reuse in an Object-Oriented System,” *Proceeding SSR '95 Proceedings of the 1995 Symposium on Software reusability,* vol. 20, no. SI, pp. 259-262, 1995.

[16] J. Kaur dan S. Singh, “Neural Network based Refactoring Area Identification in Software System with Object Oriented Metrics,” *Indian Journal of Science and Technology,* vol. 9, no. 10, pp. 1-8, 2016.

[17] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia dan D. Poshyvanyk, “Detecting Bad Smells in Source Code Using Change History Information,” *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference,* pp. 268 - 278, 2013.

[18] S. R. Chidamber dan C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering,* vol. 20, no. 6, pp. 476 - 493, 1994.

[19] S. Yu dan S. Zhou, “A Survey on Metric of Software Complexity,” *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference,* pp. 352 - 356, 2010.

[20] T. J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering,* vol. 2, no. 4, pp. 308 - 320, 1976.

[21] Istiadi, E. B. Sulistiarini dan G. D. Putra, “Enhancing Online Expert System Consultation Service with Short Message Service Interface,” *2014 I st International Conference on Infonnation Technology, Computer and Electrical Engineering (ICITACEE),* pp. 266-271, 2014.