BAB 2
LANDASAN TEORI

2.1 *Software Maintainability*

Pada standar *ISO/IEC 25010* [3], *software quality* dibagi menjadi 8 karakteristik utama yaitu *maintainability*, *functional suitability*, *performance efficiency*, *compatibility*, *usability*, *reliability*, *security*, dan *portability*. Karakteristik *maintainability* dibuat untuk mendapatkan jawaban atas pertanyaan "Apakah sebuah sistem dapat dengan mudah diubah dan dapat dengan mudah beradaptasi pada lingkungan yang baru?", jika jawabannya iya maka sistem dapat dikategorikan sebagai sistem yang *maintainable* [4].

2.1.1 Faktor- Faktor *Maintainability*

Pada standar *ISO/IEC 25010* [3], *Maintainability* sendiri dibagi lagi menjadi 5 sub-faktor yang berbeda, faktor faktor inilah yang mempengaruhi tingkat *maintainability* suatu perangkat lunak, berikut 5 sub-faktor yang dimaksud [4];

1. ***Modularity***

   Faktor ini mempengaruhi sejauh mana suatu sistem yang dibangun dari modul-modul code yang terpisah dapat dimodifikasi tanpa adanya efek terhadap komponen kode yang lain.

2. ***Reusability***

   Faktor ini mempengaruhi sejauh mana suatu komponen kode dapat dimanfaatkan dalam lebih dari satu sistem yang berbeda.

3. ***Analysability***

   Faktor ini mempengaruhi keefektifan suatu kode untuk dianalisis dengan tujuan modifikasi terhadap sistem yang ada.

4. ***Modifiability***

Faktor ini mempengaruhi sejauh mana suatu sistem dapat dimodifikasi secara efektif dan efisien tanpa mengorbankan kualitas dari sistem itu sendiri.

5. ***Testability***

   Faktor ini mempengaruhi sejauh mana tingkat keefektifan kriteria sebuah tes yang telah dirancang pada sebuah sistem dapat dinilai apakah kriteria tes telah terpenuhi.

2.1.2 *Software Maintainability Measurement* Dan Metrik

Untuk melakukan pengukuran terhadap *maintainability* suatu perangkat lunak, kita perlu mengumpulkan beberapa metrik yang memudahkan kita dalam mengukur *maintainability* suatu perangkat lunak [4]. Berikut merupakan beberapa metrik yang sering digunakan untuk kualitas suatu perangkat lunak;

Tabel 2.1 Metrik-Metrik Kualitas Perangkat Lunak

Metrik	Deskripsi
Maintainability Index (MI)	Metrik ini mengukur tingkat *maintainability* suatu perangkat lunak.
Cyclomatic Complexity (CC)	Metrik ini mengukur tingkat stabilitas dari kode yang dibuat.
Halstead's Volume (HV)	Metrik ini mengukur implementasi suatu algoritma pada sebuah kode.
Percent of *Comment*s (perCOM)	Metrik ini menentukan ukuran kuantitatif dari *comment* yang ada pada kode sumber.
Weighted *Function*s Per *Class* (WMC)	Metrik ini mengukur kompleksitas suatu *class*.
Depth Of Inheritance Tree (DIT)	Metrik ini mengukur panjang maksimum jumlah *node* dari suatu *root* pada *class hierarchy*.
Number Of Children (NOC)	Metrik ini mengukur banyaknya sub*class* dari suatu *class*.
Response For A *Class* (RFC)	Metrik ini mengukur jumlah total fungsi yang dapat dipanggil untuk merespons pesan yang telah diterima oleh suatu object *class*.

Metrik	Deskripsi
Lack Of Cohesion In *Function*s (LCOM)	Metrik ini mengukur jumlah fungsi yang tidak saling terhubung dalam sebuah *class* dan mewakili bagian yang tidak memiliki kohesi.
Data Abstraction Coupling (DAC)	Metrik ini mengukur kompleksitas coupling yang disebabkan oleh data berjenis *abstract*.
Number Of *Function*s (NOM)	Metrik ini mengukur jumlah total fungsi dalam suatu kelas termasuk fungsi *public*, *private* maupun *protected*.
Weighted *Function*s Per *Class* (WMC)	Metrik ini mengukur jumlah total dari kompleksitas fungsi yang ada pada suatu *class*.
Message Passing Coupling (MPC)	Metrik ini mengukur jumlah pemanggilan fungsi yang didefinisikan di dalam suatu *class* dan fungsi dari *class* lain
Lines Of Code (LOC)	Metrik ini mengukur jumlah total baris kode yang ada dalam suatu *class*.

2.1.2.1 Perhitungan *Maintainability Index*

Maintainability index pertama kali diperkenalkan oleh Paul Oman dan Jack Hagemeister pada tahun 1992. *Index* ini telah digunakan secara luas sebagai metrik untuk mengukur tingkat *Maintainability* suatu perangkat lunak [5]. Berikut beberapa rumus yang dapat digunakan untuk mengukur *maintainability index* suatu perangkat lunak;

1. **Paul Oman Dan Jack Hagemeister** [4]

$$MI = 171 - 5.2 * ln(V) - 0.23 * CC - 16.2 * ln(LoC) \quad \text{(II.1)}$$

2. **The Software Engineering Institute** [4]

$$MI = \left(\left(171 - \left(5.2 * \ln(V)\right) - (0.23 * CC)\text{-}(16.2 * \ln(LoC)\right) * \frac{100}{171} + 50 * \sin\left(\sqrt{2.4 * CM}\right)\right) \quad \text{(II.2)}$$

3. **Visual Studio** [4]

$$MI = MAX\left(0, \left(171 - 5.2 * \ln(V) - 0.23 * (CC) - 16.2 * \ln(LoC)\right) * \frac{100}{171}\right) \quad \text{(II.3)}$$

Dimana :

MI = *Maintainability Index*

V = Halstead Volume

CC = Cyclomatic Complexity

LoC = Line Of Code

CM = Percent Of *Comment*

2.1.2.2 Perhitungan Halstead Volume

Halstead Volume merupakan suatu metrik yang menunjukan kompleksitas suatu module berdasarkan jumlah *operator* dan *operand* yang ada pada module tersebut. Makin tinggi *Halstead Volume* maka suatu kode akan menjadi lebih kompleks dan menghambat proses pengembangan. Untuk menghitung metrik ini kita dapat menggunakan rumus berikut;

$$V = N * log2(n) \quad \text{(II.4)}$$

Dimana;

V = *Halstead Volume*

N = *Program length*

n = *Program Vocabulary*

Untuk menghitung *Program length* dan *Program Vocabulary* sendiri dapat menggunakan rumus berikut;

$$\mathrm{N} = \mathrm{N1} + \mathrm{N2} \quad \text{(II.5)}$$

$$\mathrm{n} = \mathrm{n1} + \mathrm{n2} \quad \text{(II.6)}$$

Dimana;

N = *Program length*

N1 = Jumlah *total operator*

N2 = Jumlah total *operand*

n = *Program Vocabulary*

n1 = Jumlah *operator* unik

n2 = Jumlah *operand* unik

2.1.2.3 Perhitungan Cyclomatic Complexity

Cyclomatic Complexity merupakan metrik yang diperkenalkan oleh *McCabe*. Metrik ini menunjukan kompleksitas *structural* suatu module dengan mempertimbangkan aliran kontrol pada suatu modul. Makin tinggi kompleksitas suatu kode semakin sulit pula kode tersebut dapat di *maintenence* oleh pengembang [6]. Untuk menghitung metrik ini kita dapat menggunakan rumus berikut;

$$CC = E - N + 2P \tag{II.7}$$

Dimana;

CC = Cyclomatic Complexity

E = Jumlah *edge* pada suatu *flow graph*

N = Jumlah *node* pada suatu *flow graph*

P = Jumlah Keluaran

2.1.2.4 *Tools* Pengukuran Metrik Perangkat Lunak

Selain menggunakan perhitungan secara manual, kita dapat menggunakan alat untuk membantu dalam melakukan perhitungan Maintainbility. Berikut beberapa contoh alat yang dapat digunakan pengembang dalam menghitung metrik kualitas suatu perangkat lunak [5];

Tabel 2.2 Tool-Tool Pengukuran Metrik Perangkat Lunak

Tools	Metriks	Deskripsi
PhpMetric	Dapat mengidentifkasi 11 tipe metrik *maintainability*.	Alat ini digunakan untuk menganalisis tiap metrik-metrik yang pada sebuah sistem. Alat ini hanya dikhususkan untuk Bahasa pemrograman PHP.
SourceMeter	Dapat mengidentifikasi lebih dari 50 tipe metrik *maintainability*.	Alat ini memungkinkan untuk menemukan titik lemah dari suatu sistem yang sedang dikembangkan hanya dengan melihat kode sumbernya saja dan juga mendukung lebih dari 5 bahasa pemrograman yang berbeda.
IntelliJ IDEA	Dapat	Merupakan suatu *plugin* IDE yang dapat

Tools	Metriks	Deskripsi
	mengidentifikasi 24 metrik *maintainability*.	membantu dalam mengkalkulasi metrik-metrik yang ada didalam perangkat lunak.

2.1.2.5 Tool PhpMetric

PhpMetrics merupakan suatu alat statis analisis yang digunakan untuk menganalisis metrik-metrik kualitas perangkat lunak yang dibangun menggunakan bahasa pemrograman PHP. Metrik-metrik yang dapat dianalisis oleh *PhpMetric* ada banyak mulai dari metrik *Maintainanbility Index*, *Cyclomatic Complexity*, *Halstead Volume*, *Line Of Code*, dll [7]. *PhpMetric* dapat didownload menggunakan *dependency management tool "Composer"*. *PhpMetric* akan memberikan penilaian hasil perhitungan setiap metrik dalam bentuk HTML yang dapat dilihat pengguna menggunakan *browser* tanpa adannya koneksi internet [7].

2.2 *Clean Code*

Clean Code merupakan prinsip/metode yang dipopulerkan oleh Robert C. Martin dalam bukunya yang berjudul "*Clean Code A Handbook of Agile Software Craftsmanship*". Dalam bukunya Robert menjelaskan secara lengkap mengenai *Clean Code*, beliau juga mengutip beberapa pendapat para pengembang berpengalaman mengenai definisi dari *Clean Code* diantaranya :

1. Menurut Grady Booch, *Clean Code* adalah kode yang sederhana dan tepat sasaran, "kode yang bersih tidak pernah mengaburkan maksud perancang melainkan kode yang ada penuh dengan abstraksi dan juga kontrol yang jelas" [1].
2. Menurut Bjarne Stroustrup, *Clean Code* didefinisikan dengan menggunakan kata "elegan" dimana beliau berpendapat bahwa "logika yang ada pada sebuah kode haruslah jelas sehingga akan lebih mudah sebuah *bug* dapat ditemukan, ketergantungan yang ada di suatu kode juga haruslah seminimal mungkin sehingga lebih mudah untuk *maintenance*" [1].

3. Menurut “Big” Dave Thomas, *Clean Code* haruslah bisa dibaca dan di kembangkan oleh pengembang lain selain pengembang utama. *Clean Code* haruslah memiliki *unit test*, nama yang memiliki arti yang jelas, memiliki ketergantungan yang minimal dan juga kode yang ada haruslah menyediakan satu cara saja dibanding banyak cara untuk melakukan satu hal [1].

2.2.1 Faktor Clean Code

Ada banyak faktor yang perlu diperhatikan dalam mengimplementasikan *Clean Code* pada kode sumber yang dibuat diantaranya adalah;

1. ***Meaningful Name***

   Dalam menulis sebuah code program “nama” merupakan hal yang paling sering ditemui oleh pengembang. Pengembang perlu memberi nama terhadap setiap hal yang ada di kode program seperti variabel, fungsi, *argument*, *class*, dll. Karena pengembang selalu memberi nama pada kode mereka maka pengembang perlu memikirkan nama dengan seksama agar pengembang tidak bingung terhadap nama yang telah diberikan [2].

2. ***Function***

   Dalam pemrograman berbasis *object function* merupakan hal yang pasti akan dijumpai pengembang. *Function* dapat mempermudah pengembang dalam membaca dan mengorganisir kode mereka sehingga faktor *function* merupakan faktor yang perlu diperhatikan dalam penerapan *Clean Code* [2].

3. ***Comment***

   Comment diperlukan dalam memberikan informasi mengenai kode yang ditulis oleh pengembang. Dengan menulis komentar yang baik, pengembang dapat dengan mudah mengerti apa yang dimaksud oleh pengembang lain, dan sebaliknya apabila menulis komentar yang buruk maka pengembang malah akan lebih sulit untuk memahami kode yang ditulis oleh pengembang lain [2].

4. ***Formating***

Pemformatan kode merupakan hal yang penting, pemformatan suatu kode adalah tentang komunikasi antar pengembang. Dengan adanya format yang baik dan konsisten para pengembang dapat dengan mudah saling berkomunikasi karena kode yang ditulis akan jauh lebih mudah dibaca jika format penulisan yang digunakan baik dan konsisten [2].

5. ***Object And Data Structure***

   Dalam suatu kode program pasti ada variable/data yang tercantum didalammnya. Pengembang perlu untuk memperhatikan bagaimana data pada suatu kode program diorganisasi. Dengan adannya pengorganisasian data yang baik code yang ditulis pengembang dapat lebih mudah untuk diubah dan dikelola [2]

6. ***Error Handling***

   Error Handling merupakan salah satu hal yang perlu diperhatikan seorang pengembang dalam menulis kode program. Seroang pengembang yang menulis *Error Handling* dengan benar apabila terjadi *error* pengembang dapat dengan mudah mengerti *error* yang didapatinnya dan dapat dengan mudah memperbaikinnya sehingga waktu pengembangan dapat dikurangi [2].

7. ***Class***

   Sama halnya dengan *function*, *class* merupakan salah satu hal yang tidak bisa dihindari dalam pemrograman berbasis objek. Dengan adanya *class* pengembang dapat dengan mudah mengorganisasi kode mereka buat sehingga faktor *class* merupakan faktor yang perlu diperhatikan dalam penerapan *Clean Code* [2].

2.2.2 *Code Smells*

Code Smells bukanlah suatu *error* yang ada pada sebuah kode sumber akan tetapi *Code Smells* adalah suatu kelemahan dari sebuah desain sistem perangkat lunak yang akan menghambat pengembangan suatu suatu perangkat lunak atau bisa juga meningkatkan ancaman pada sistem pada masa mendatang. *Code Smell* juga dikenal sebagai “*Bad Smells*”, “*Anti Pattern*”, “*Code Anomaly*”

ataupun "*Design Flaws*" [8]. Pada tahun 2003, Mantyla mengkategorikan *Code Smell* yang ada pada buku Martin Fowler menjadi 7 kategori yang berbeda berdasarkan karakteristiknya [8]. Berikut merupakan ketujuh kategori yang dimaksud;

Tabel 2.3 Pemetaan *Code Smell* Berdasar Karakteristiknya

No	**Kategori**	**Deskripsi**	***Code Smells***
1	*Bloaters*	*Bloaters* merupakan sebuah *class* atau *function* yang terlalu besar sehingga sulit untuk dikelola	*Large class, long parameter list, data clumps, long function, dan primitive obsession.*
2	*Object Orientation abusers*	*Code smell* jenis ini terkait dengan keadaan dimana sistem tidak dapat melakukan perubahan secara langsung	*Alternative classes with different interfaces, temporary field, switch statements, parallel inheritance hierarchies, refused bequest*
3	*Change preventers*	*Code Smell* jenis ini terkait kode yang menghambat proses modifikasi suatu perangkat lunak	*Shotgun surgery, divergent change*
4	*Dispensables*	*Code Smell* jenis ini terkait dengan sesuatu yang tidak perlu atau tidak berguna sehingga dengan menghapusnya dapat membuat kode yang ada lebih mudah dimengerti	*Data class, speculative generality, duplicate code, lazy class*
5	*Encapsulators*	*Code Smell* jenis ini terkait dengan enkapsulasi suatu data maupun mekanisme komunikasi pada sebuah sistem	*Middle man, message chains*

6	*Couplers*	*Code Smell* jenis ini terkait dengan banyaknya coupling dalam sebuah code program	*Inappropriate intimacy, feature envy*
7	*Others*	*Code Smell* jenis ini merupakan *Code Smell* yang tidak cocok dari salah satu kategori yang lain	*Comments, incomplete library class*

Menurut Kessentini et al, terdapat 7 teknik yang dapat digunakan untuk mendeteksi *Code Smell* yaitu *Manual, metrik-based detection approach, symptom-based approach, probabilistic-based approach, visualization-based approach,* dan *search-based approach*. Untuk sistem berukuran kecil penggunaan teknik manual menjadi salah satu cara terbaik untuk mendeteksi *Code Smell* akan tetapi untuk sistem yang lebih besar kita dapat menggunakan alat seperti *InCode, JNOSE,* dan *PRODEOOS* untuk membantu pengembang dalam mendeteksi *Code Smell* [8].

2.3 *Design Pattern*

Design Pattern merupakan suatu solusi yang digunakan untuk masalah umum yang sering terjadi pada desain perangkat lunak [9]. *Design Pattern* seperti halnya cetak biru yang bisa dipakai *programmer* ketika ingin memecahkan suatu masalah terkait pengembangan perangkat lunak. *Design Pattern* dan Algoritma merupakan konsep yang mirip tapi tidak sama, Algoritma merupakan beberapa langkah yang perlu diikuti untuk memecahkan suatu masalah sedangkan *Design Pattern* merupakan suatu cetak biru yang dapat dipakai kapan saja dan urutan penerapannya tergantung *programmer* yang mengimplementasikannya.

2.3.1 Klasifikasi *Design Pattern*

Design Pattern dibagi 2 jenis utama berdasarkan tingkat kegunaannya dimana untuk tingkat rendah atau sering disebut *idiom* merupakan *Design Pattern* yang biasannya berlaku hanya untuk satu bahasa pemrogramman sedangkan *Design Pattern* tingkat tinggi atau yang sering disebut pola arsitektur merupakan

Design Pattern yang dapat digunakan pada hampir semua bahasa pemrograman dan dapat digunakan untuk merancang arsitektur suatu perangkat lunak [9]. Selain itu *Design Pattern* juga dapat dikategorikan berdasarkan maksud dan tujuan digunakannya dan berikut merupakan beberapa kategori yang dimaksud;

1. ***Creational Pattern***

   Kategori ini digunakan untuk pembuatan suatu objek kategori ini dapat meningkatkan fleksibilitas dan reusabilitas pada suatu kode program [9].

2. ***Structural Pattern***

   Kategori ini digunakan untuk membuat suatu objek dan *class* dapat diorganisasikan menjadi struktur yang lebih besar dengan menjaga flexsibilitas dan efisiensi dari struktur yang ada [9].

3. ***Behavioral Pattern***

   Kategori ini digunakan untuk menjaga efektifitas komunikasi dan pembagian tanggung jawab antar objek/*class* [9].

2.3.2 *Design Pattern* Dan Kegunaannya

Berikut merupakan beberapa *Design Pattern* yang sering digunakan beserta penjelasan fungsinya masing-masing *pattern*;

1. ***Creational Pattern***

Tabel 2.4 *Creational Desing Pattern*

No	*Design Pattern*	Fungsi
1	*Singleton*	Digunakan agar suatu *class* hanya memiliki satu buah *instance* yang dapat digunakan sebagai akses point global untuk semua *class* yang ada [9].
2	*Prototype*	Digunakan untuk membuat suatu objek yang memiliki *property* yang sama (*Clone Object*) [9]**.**
3	*Factory Method*	Menyediakan suatu *interface* yang digunakan untuk membuat objek berdasarkan dari *superclass* akan tetapi *class* turunannya bisa mengatur tipe apa yang ingin dibuat [9].
4	*Abstract Factory*	Digunakan untuk membuat sebuah *object families* tanpa

No	*Design Pattern*	Fungsi
		spesifik terhadap *class* turunannya [9].
5	*Builder*	Digunakan untuk membuat suatu objek yang kompleks yang memiliki tipe yang berbeda menggunakan kode konstruksi yang sama [9]**.**

2. Structural Pattern

Tabel 2.5 *Structural Design Pattern*

No	*Design Pattern*	Kegunaan
1	*Adapter*	Digunakan untuk membuat suatu objek yang memiliki *interface* yang berbeda saling berkolaborasi [9].
2	*Bridge*	Digunakan untuk memisah suatu *class* yang besar menjadi 2 hirarki yang berbeda yaitu *abstraksi* dan implementasi [9].
3	*Facade*	Digunakan untuk menyederhanakan suatu *interface* suatu *library*, *framework* atau suatu *class* yang kompleks [9].
4	*Repository*	Merupakan spesialisasi dari F*açade Pattern* yang digunakan untuk memisahkan algorimta akses data dan juga logika bisnis [9].
5	*Decocator*	Digunakan untuk menambah suatu *behavior*/method pada suatu objek dengan cara menempatkan objek tersebut pada suatu objek khusus yang bertugas sebagai pembungkus [9].

3. Behavioral Pattern

Tabel 2.6 *Behavioral Design Pattern*

No	*Design Pattern*	Kegunaan
1	*Observer*	Digunakan untuk membuat suatu *system subscription* yang digunakan untuk menginformasikan kumpulan objek akan suatu *event* yang terjadi [9].
2	*Mediator*	Digunakan untuk memediasi beberapa objek agar dapat

No	*Design Pattern*	Kegunaan
		berkomunikasi pada suatu tempat khusus untuk mengurangi keterikatan antar objek [9].
3	*Strategy*	Digunakan untuk memisah beberapa algoritma yang mirip ke beberapa *class* sehingga dapat digunakan dengan mudah [9].
4	*Template Method*	Digunakan untuk membuat kerangka algoritma pada *superclass* agar *class* turunannya dapat menggunakannya [9].
5	*Iterator*	Digunakan untuk melihat setiap elemen pada suatu *collection* (*list, array, stack,* dll) tanpa membongkar representasi dari elemen tersebut [9].

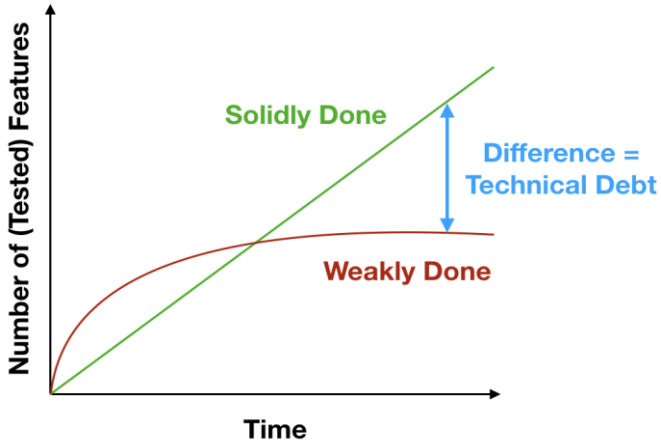
2.4 *Technical Debt*

Istilah *Technical Debt* pertama kali diciptakan oleh Ward Cunningham untuk menggambarkan efek negatif jangka panjang dari sebuah kode yang belum matang yang terjadi selama *software development lifecycle* (SDLC). Cunningham menggambarkan istilah *Technical Debt* sebagai sebuah hutang dan bunga yang harus dibayar oleh pengembang. Setiap pengembang menulis sebuah kode yang buruk akan dianggap sebagai hutang yang perlu dibayar (Diperbaiki) jika hutang itu dibayar dengan cepat maka bunga yang didapat oleh pengembang akan lebih sedikit sedangkan jika dibiarkan bunga akan membengkak dan perangkat lunak akan lebih lama untuk dikembangkan [10].

2.4.1 Dampak *Technical Debt*

Dalam industri *software development*, *Technical Debt* (TD) diakui sebagai masalah yang sangat serius saat ini, jika dibiarkan TD dapat menyebabkan pembekakan biaya produksi, hal ini disebabkan karena adanya biaya *maintenance* yang tinggi dan ketidakmampuan suatu perangkat lunak dalam menambahkan suatu fitur dan bahkan pada titik krisis tertentu TD dapat menyebabkan suatu perangkat lunak perlu dilakukan *refactoring* secara menyeluruh kepada sistem yang ada [10].

Dalam sebuah studi yang dilakukan oleh Besker, T terhadap 43 pengembang, yang dilakukan selama 2 minggu dapat dibuktikan bahwa TD menghambat pengembang dalam mengembangkan perangkat lunak mereka, Pengembang melaporkan bahwa rata-rata 23% dari semua waktu kerja mereka terbuang sia sia karena TD. Waktu yang terbuang ini disebabkan karena selama adanya TD, pengembang biasanya harus melakukan pengujian tambahan, analisis kode sumber serta melakukan *refactoring* ulang terhadap kode yang ada [10].



Gambar 2.1 Dampak *Technical Debth*

2.4.2 *Tools* Analisis *Technical Debt*

Terdapat beberapa *tools* yang dapat digunakan untuk menganalis TD. Berdasarkan wawancara yang dilakukan pada sebuah penitian ditemukan bahwa sebagian besar *tools* yang digunakan responden untuk melakukan analisis adalah dengan menggunakan *project management tool* seperti *Jira*, *Readmine*, dan *Team Foundation Server* [11]. Selain menggunakan *project management tool*, pengembang juga dapat melakukan analisis dengan menggunakan *static code analysis tool* contohnya *SonarQube*, *Codescene*, dan *Raxis*.

Menurut penelitian, alat alat yang digunakan saat ini untuk melakukan analisis TD belumlah mencapai hasil yang memuaskan, masih banyak potensi metrik dan teknik yang belum dimanfaatkan yang akan berpotensi dapat meningkatkan estimasi konsep analisis TD. Dan dengan banyaknya penelitian TD

diharapkan dapat mendorong *development* produk perangkat lunak berkualitas tinggi [12].

2.5 *Refactoring*

Refactoring merupakan sebuah proses untuk meningkatkan struktur *internal* sebuah perangkat lunak tanpa mengubah perilaku *external* dari kode yang ada [13]. Dengan melakukan *Refactoring* pengembang dapat membersihkan kode yang ditulis dari kemungkinan adanya *bug*. Dalam mengembangkan suatu perangkat lunak, pengembang biasanya mendesain terlebih dahulu kode yang ingin dibuat kemudian melakukan *coding* setelahnya akan tetapi dengan menggunakan *refactoring*, pengembang dapat memulainya tanpa melakukan *design* terlebih dahulu dan kemudian apabila kode yang ditulis memiliki desain yang jelek, pengembang dapat mengubah desain yang ada agar menjadi lebih baik dengan menggunakan *refactoring* [13].

2.5.1 Tipe *Refactoring*

Dalam penerapannya *refactor* dibagi menjadi 2 tipe yang pertama, kedua tipe itu adalah;

1. ***Proactive Refactoring***

   Refactoring jenis ini digunakan ketika didalam kode belum ditemukan *Code Smells* akan tetapi akan segera terbentuk sehingga pengembang akan secara aktif melakukan *refactor* bahkan ketika tidak adanya masalah yang ada didalam kode [14].

2. ***Reactive Refactoring***

   Refactoring jenis ini digunakan ketika kode yang ada sudah memiliki *Code Smells* yang harus segera diperbaiki oleh pengembang untuk membersihkan kode sumber yang ada [14].

Menurut Sae-Lim, Hayashi dan Saeki, tidaklah praktis untuk melakukan *refactoring* pada setiap *class* dan module yang ada pada sebuah kode oleh sebab itu pengembang dapat menghitung *module decay index* (MDI) untuk menentukan

apakah sebuah module akan menciptakan *Code Smells*. Dengan menggunakan tipe *refactoring proactive* pengembang dapat memperkirakan modul mana saja yang akan menciptakan *Code Smell* dan memperbaikinya sebelum masalah muncul [14].

2.5.2 Teknik *Refactoring*

Setelah dipopulerkan oleh Fowler dalam bukunya berjudul *Refactoring: Improving the Design of Existing Code*, teknik *refactoring* terus berkembang setiap tahunnya. Didalam buku martin fowler sendiri terdapat 68 teknik *refactoring* yang dibagi menjadi 6 kategori dan menurut sebuah penelitian menggunakan sampel dari 20 teknik *refactoring* yang paling sering digunakan pada riset berdasar studi literatur, peneliti melakukan survei terhadap pengembang apakah teknik-teknik *refactoring* yang ada memang benar sering dipakai pada praktek dilapangan dan hasilnya adalah [15];

Tabel 2.7 Teknik-Teknik *Refactoring* Yang Sering Digunakan

No	Teknik	Setuju (%)	Tidak Setuju (%)
1	*Add Parameter*	50,0%	50,0%
2	*Collapse Hierarchy*	21,9%	78,1%
3	*Encapsulate Field*	93,75%	6,25%
4	*Extract Class*	90,6%	9,4%
5	*Extract Subclass*	90,6%	9,4%
6	*Extract Superclass*	62,5%	37,5%
7	*Extract Function*	87,5%	12,5%
8	*Inline Class*	54,8%	45,2%
9	*Inline Function*	58,1%	41,9%
10	*Introduce Null Object*	13,3%	86,7%

No	Teknik	Setuju (%)	Tidak Setuju (%)
11	*Move Field*	86,7%	13,3%
12	*Move Function*	87,1%	12,9%
13	*Pull Up Field*	96,7%	3,3%
14	*Push Down Field*	93,3%	6,7%
15	*Pull Up Function*	96,7%	3,3%
16	*Push Down Function*	96,7%	3,3%
17	*Rename Function*	93,75%	6,25%
18	*Remove Parameter*	41,9%	58,1%
19	*Replace Function With Function Object*	9,68%	90,32%
20	*Replace Conditional With Polymorphism*	12,9%	87,1%