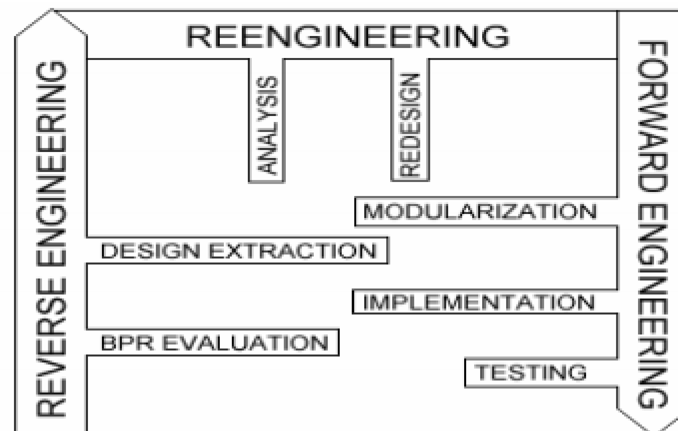
BAB 2
LANDASAN TEORI

2.1 Software *Reengineering*

Software *reengineering* adalah sebuah proses yang dilakukan untuk melakukan implementasi ulang pada sistem yang sudah dengan menggunakan metode dan teknik yang berbeda dengan tujuan untuk *redesign* dan *reshape* sebuah sistem agar menjadi lebih terpelihara dan modern [3]. Berdasarkan "*Reengineering and Reengineering Patterns"* ada beberapa fase yang terjadi Ketika melakukan proses software *reengineering*:



Gambar 1 Gambaran Fase umum dalam Proses *Reengineering*

2.1.1 Fase pada Proses Reengineering

2.1.1.1 Reverse Engineering

Reverse Engineering adalah proses yang dilakukan pada awal reengineering sebuah aplikasi, tujuan umu dari proses ini adalah untuk menggali informasi didalam sistem seperti informasi design dalam *source code*. Didalam proses reverse engineering ada beberapa sub proses lainnya seperti :

1. BPR (Business Process Reengineering) Evaluation

   Pada proses ini dilakukan analisis pada proses bisnis yang sedang berjalan, pada proses ini akan ditentukan apa saja keperluan bisnis yang ada,

tujuan bisnis yang ada, dan juga apakah ada proses bisnis yang akan diubah [3].

2.1.1.2 Design Extraction

Proses ini dilakukan dengan tujuan untuk mengekstrak informasi yang ada pada aplikasi sebelumnya, seperti komponen komponen atau artefak dari source code sebelumny, ini dilakukan untuk membuat gambaran abstrak dari design yang ada seeblumnya [9].

2.1.1.3 Reengineering

Pada bagian ini dilakukan analisis dan *redisgn* sesuai dengan kebutuhan dari komponen *source code* yang ditemukkan pada proses *Reverse Engineering.* Beberapa proses yang ada dalam tahapan ini seperti :

1. Analysis

   Proses ini bertujuan untuk melakukan analisis terhadap keperluan yang sudah didapat sebelumnya, pada proses ini ada kemungkinan muncul keperluan fungsionalitas yang baru [3]

2. Redesign

   Setelah seluruh keperluan fungsionalitas didefinisikan ditahapan analisis sebelumnya, akan dilakukan proses redesign untuk menggabungkan sistem yang lama dengan keperluan fungsionalitas yang baru [3]

2.1.1.4 Forward Engineering

Proses terakhir ialah Forward Engineering, proses ini adalah proses yang umum dilakukan pada proses software engineering pada umumnya, pada proses ini dilakukan implementasi dari hasil analisis dan redesign yang sudah dilakukan sebelumnya. Proses forward engineering memiliki 3 tahapan yaitu :

1. Modularization

    Proses modularization bertujuan untuk memecah belah aplikasi menjadi bentuk *module* dan *classes.* Hal ini bertujuan untuk menjaga reusability dari sistem. Proses modularization akan menghasilkan program yang memiliki ukuran lebih kecil dan juga dapat mengurangi kompleksitas program [3]

2. Implementation

    Proses implementation adalah proses implementasi kode program. Dalam proses implementasi penting untuk mengikuti sebuah aturan atau konsep, untuk menjaga *maintainability* dan *reusability* dari kode tersebut [3].

3. Testing

    Testing dilakukan untuk menemukan kesalahan atau *error* yang terjadi pada kode program, selain itu testing juga ditujukan untuk melakukan optimisasi dari kode yang sudah berjalan dengan baik, Ketika melakukan testing akan ada skenario skenario pengujian yang dilakukan untuk menguji aplikasi yang ada [3].

2.1.2 Enhanced Reengineering

Enhanced reengineering adalah salah satu mekanisme proses software reengineering, proses Enchaned reengineering menggunakan berbagai metode abstraksi untk mengubah software lama menjadi software baru. Mekanisme Enhanced reengineering menggunakan kedua proses *reverse engineering* dan juga *forward engineering* [8]. Proses Enhanced Reengineering memiliki 5 fase utama, yaitu :

1. Feasibility Study and Requirements

    Pada fase ini dilakukan analisis kelayakan dan persyaratan yang ada pada sistem sebelumnya, setelah analisis dilakukan hasil nya akan disesuaikan dengan keperluan user. Hasil akhir dari proses ini ialah sebuah Software Requirement Specification (SRS) yang berisikan spesifikasi dari keperluan sistem [8].

2. Restructured System Requirements Specification

Pada proses ini akan dibentuk struktur sistem dan dari Software Requirement Specification yang sudah dibuat sebelumnya, hasil akhir dari proses ini adalah sebuah cetak biru yang menggambarkan proses reengineering yang sudah dilakukan, pada proses juga akan dilakukan pembandingan sistem lama dengan sistem yang baru dengan menggunakan software requirement specification yang sudah dibuat [8].

3. Design To Code

Dari redesign document yang sudah dibuat pada tahap sebelumnya, biasanya akan dilakukan perbandingan algoritma pada sistem lama dan yang akan dibuat, bila ditemukkan sebuah algoritma atau kode yang sudah tidak lagi cocok digunakan akan dilakukan pergantian algoritma ke algoritma yang lebih modern [8].

4. Comparison of Existing and proposed Functionalities

Pada fase ini dilkukan perbandingan antara sistem yang lama dengan sistem yang baru, performa dari fungsionalitas kedua sistem akan dibandingkan, dan di evaluasi dengan menggunakan metric seperti *running time*, penggunaan memory dan konfigurasi sistem [8].

5. Implementation

Pada fase terakhir Enhanced reengineering, dilakukan implementasi pembuatan sistem yang baru, pada proses implementasi bagian-bagian spesifik dapat diganti dengan menggunakan hasil dari proses sebelumnya [8].

2.2 Maintainability

Maintainability adalah salah satu faktor pada Software Quality, dimana maintainability berfokus pada kemudahan sebuah aplikasi untuk dipelihara [7]. Seperti :

1. Memperbaiki kerusakan
2. Menemukan kebutuhan baru
3. Membuat pemeiliharan selanjutnya lebih mudah
4. Mengatasi lingkungan yang berubah.

2.2.1 Maintainability Index

Tingkat *maintainability* sebuah aplikasi dapat diukur dengan menggunakan *Maintainability Index* (MI), Maintainability index menggabungkan sejumlah metric tradisional seperti *Average Halstead Volume per Module*, *Cyclomatic Complexity,* banyak nya *lines of codes (LOC)* dan juga rasio komentar pada *source code* untuk menghitung tingkat *maintainability* pada sebuah aplikasi [10][11]. Maintainability index memiliki formula sebagai berikut :

$$MI = 171 - 5.2 \ln V - 0.23G - 16.2 \ln L$$

Adapun formula versi lain yang berupa turunan dari *Visual Studio Code* [8] [9]:

$$MI = \max[0{,}100 \frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L}{171}]$$

Dari versi formula Visual Studio Code Microsoft, diberikan jarak antara 0-100 untuk menilai maintainability yang dimana semakin tinggi nilai nya semakin baik pula tingkat maintainability. Adapun pengelompokan tingkat maintainability sebagai berikut [10][12]:

Skor Maintainability	Tingkatan
0-65	Maintainability Rendah
66-85	Maintainability Menengah
86 >	Maintainability Tinggi

Dimana :

$V = Halstead\ Volume$ $L = Jumlah\ line\ of\ code\ (LOC)$

$G = Total\ Cyclo\ Matic\ Complexity$ $C = Rasio\ komentar$

2.3 Halstead Metric

Menurut Halstead, program komputer adalah implementasi dari algoritma yang dianggap sebagai kumpulan token yang dapat diklasifikasikan sebagai operator atau operan. Dengan kata lain, program dapat dianggap sebagai urutan operator dan operan terkaitnya. Semua metrik Halstead adalah fungsi dari jumlah token ini. Dengan menghitung token dan menentukan operator yang operan berdasarkan strategi penghitungan, ukuran dasar berikut dapat dikumpulkan [13] :

$$n_1 = Jumlah\ operator\ yang\ unik$$
$$n_2 = Jumlah\ Operand\ yang\ unik$$
$$N_1 = Jumlah\ total\ muncul\ nya\ operator$$
$$N_2 = Jumlah\ total\ muncul\ nya\ operand$$

Disamping operator operator diatas Halstead juga menambahkan :

$$n_1 *= Jumlah\ \ operator\ potensial$$
$$n_2 *= Jumlah\ operand\ potensial$$

Adapun metrik-metrik Halstead sebagai berikut :

1. Program Length

   Mendefinisikan Panjang dari sebuah program P.

$$N = N_1 + N_2$$

2. Program *Vocabulary*

   Mendifinisikan *vocabulary* dari sebuah program

$$n = n_1 + n_2$$

3. Program Volume

   Mendifiniskan proposi besar sebuah program dalam memori, halstead mengatakan metric ini cocok digunakan untuk mengukur besar nya implementasi sebuah algoritma. Dapat dihitung dengan menggunakan :

$$V = N * log_2 n$$

4. Potential Minimum Volume

   Metric ini mendifinisikan potensi besar minimum dari implementasi sebuah algoritma atau program, di definisikan sebagai :

$$V^* = (2 + n_2^*) log_2 (2 + n_2^*)$$

5. Program Level

   Metric ini digunakan untuk menentukan apakah sebuah program memiliki high level atau low level, semakin tinggi level program maka akan semakin mudah juga pengembangannya, di definisikan sebagai :

$$L = \frac{V^*}{V}$$

6. Program Difficulty

Metric ini digunakan untuk mengetahui kesulitan sebuah program, di definisikan sebagai :

$$D = \frac{1}{L}$$

7. Programming Effort

Ukuran ini digunakan untuk mengetahui seberapa banyak usaha yang diperlukan untuk mengubah sebuah algoritma program, di definisikan sebagai :

$$E = \frac{V}{L} = \frac{n_1 N_2 N log_2 n}{2 n_2}$$

8. Intelligence Content

Metric ini digunakan untuk mengukur ada berapa banyak konten informasi yang ada didalam sebuah program, di definisikan sebagai :

$$I = L * V$$

9. Programming Time

Programming time adalah metric yang digunakan untuk mengukur berapa lama program dibuat (T) dengan jumlah *effort* (E), di definisikan sebagai:

$$T = \frac{E}{S}$$

2.4 Cyclomatic Complexity

Cyclomatic Complexity McCabe adalah salah satu *software metric* yang digunakan untuk mengukur kompleksitas sebuah progam. *Cyclomatic Complexity* mengukur jalur indpenden linear melalui *source code* program. *Cyclomatic Complexity* dapat digunakan untuk mengukur individu fungsi, module, method ataupun kelas pada sebuah program [11].

Cyclomatic Complexity dihitung dengan menggunakan *control flow graph* dari sebuah program, Setelah graf program dibentuk Cyclomatic Complexity dapat dihitung dengan menggunakan :

$$V(G) = E - N + 2$$

Dimana :

- V(G) : Cyclomatic Complexity
- E Jumlah *edge* pada suatu program
- N Jumlah *node* pada suatu program

2.5 Golang

Gambar 2 Logo Bahasa Pemrograman Golang

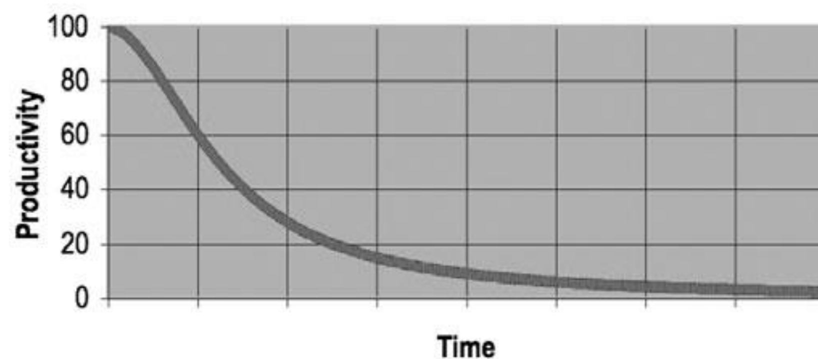
Bahasa pemrograman golang adalah bahasa pemrograman yang dikeluarkan oleh google pada tahun 2009. Golang merupakan bahasa pemrograman yang bersifat *strong typing*, *asynchronous,* dan juga berorientasi objek. Golang dibuat dengan tujuan untuk memaksimalkan potensi dari *multi-processor* yang umum adanya di era modern saat ini. Dengan tujuan itu juga golang dibangun dengan adanya modul "Goroutine" yaitu modul konkurensi bawaan dari bahasa pemrogaman Golang, berbeda dengan bahasa pemrograman lain yang tidak memiliki modul konkurensi bawaan [14].

Goroutine memiliki kelebihan dibandingkan menggunakan thread biasa, Goroutine hanya menggunakan sedikit resource dibandingkan thread, berbeda dengan thread yang Berbeda dengan bahasa pemrograman lain yang harus mengatur thread secara manual, Goroutine dapat diatur secara otomatis oleh Golang. Yang terakhir adalah Goroutine menggunakan channel untuk berkomunikasi dengan satu sama lain, hal ini dapat menghentikan masalah seperti *race condition* dan *shared memory problem* [15]

2.6 Clean Code

Clean code merupakan suatu konsep atau petunjuk penulisan struktur kode yang bersih *(Clean)*, di mana kode yang ditulis dapat dibaca oleh pengembang lain dan dapat diubah dengan mudah . *Clean code* bertujuan untuk mengatasi penurunan tingkat produktivitas pengembangan perangkat lunak akibat dari struktur kode yang berantakan seperti yang dapat dilihat pada Gambar 4 di mana waktu dapat mempengaruhi tingkat produktivitas [16].Adapun hal-hal yang dibahas dalam *Clean Code* seperti :

1. Meaningful Names
2. Clean Function
3. Clean Comment
4. Clean Formatting
5. Clean Object & Data Structure
6. Clean Error Handling
7. Clean Classes



Gambar 3 Productivity vs Time

2.6.1 Meaningful Names

Pada bagian ini, diberikan petunjuk untuk melakukan pemberian nama pada variable, *method,* fungsi, dan kelas. Tujuan pemberian nama ini adalah sehinga komponen-komponen tersebut mudah untuk dibaca dan dipahami. Berikut merupakan aturan-aturan Ketika memberikan nama :

1. Gunakan nama yang memiliki arti

Penggunaan nama yang mudah dibaca serta memiliki arti yang sesuai akan mempermudah pengembang lain dalam memahami nya. Sehingga pengembang tidak perlu mengeluarkan usaha lebih Ketika ingin memahami kegunaan kode tersebut. Berikut adalah contoh nama yang memiliki arti :

```
Buruk
var num int = 10
Baik
var numberOfEmployee int = 10
```

2. Nama yang mudah dicari

Nama yang baik adalah nama yang mudah dicari dan dikenali, hal ini akan membantu pengembang Ketika mencari sebuah kode, Berikut contoh kodenya :

```
Buruk
var d int = 20
Baik
var realDaysPerIdealDay = 4
const WORK_DAYS_PER_WEEK = 5
```

3. Menghindari mental mapping

Penggunaan variable seperti i & n sudah sangat sering ketika melakukan iterasi, namun pola tersebut membuat kebiasaan bahwa i & j adalah variable yang digunakan untuk iterasi. Ini tidak baik dibiarkan karena bisa jadi ada pengembang lain yang tidak menggunakan standar yang sama, dan ini dapat mempersulit pemahaman terhadap kode.

```
Buruk
if f == true {
doStuff()
}
Baik
if conditionFlag == true {
doStuff()
}
```

4. Kata benda untuk *class*/modul

Dalam melakukan pemberian nama terhadap class / modul, disarankan untuk menggunakan kata benda dan menghindari penggunaan kata kerja. Hal tersebut dikarenakan umumnya *class*/modul biasanya mewakili sebuah entitas.

5. Kata kerja untuk *method*/fungsi

Penamaan *method/*fungsi disarankan untuk menggunakan kata kerja, hal ini bertujuan untuk mendeskripsikan apa yang *method*/fungsi tersebut lakukan, Untuk mempermudah penulisan ini dapat juga digunakan prefix seperti *set, get*, dan *is* untuk memperjelas makna dari fungsi. Berikut adalah contoh kode yang baik:

```
func getUserData(){
    fetchToDatabase()
}
```

6. Hindari penggunaan konteks yang tidak diperlukan

   Ketika menuliskan sebuah modul, sering kali kita menuliskan nama objek berulang, seperti class Car dengan attribute carColor, hal tersebut tidak baik untuk dilakukan karena bersifat repetitive dan tidak memiliki makna, dibandingkan menggunakan carColor lebih baik hanya menggunakan color.

2.6.2 *Clean Function*

Pada konsep ini terdapat petunjuk dan aturan untuk menulis *method/*fungsi yang bersih agar dapat lebih mudah dipahami oleh pengembang. Berikut merupakan aturan-aturan dalam pembuatan *clean function :*

1. Hindari penulisan *method*/fungsi yang panjang

   Ketika menulis *method*/fungsi sering kali ditemukkan fungsi yang melakukan banyak hal sekaligus, hal ini tidak baik dilakukan karena hal ini dapat menyebabkan sulitnya kode untuk ditelaah. Untuk menulis sebuah *clean function* ada baiknya untuk memcah fungsi menjadi bagian yang lebih kecil sehingga lebih mudah ditelaah dan dibenahi.

2. Hindari penggunaan Flag sebagai parameter

   Penggunaan *flag* sebagai parameter dapat menandakan *method*/fungsi yang dibangun memiliki pekerjaan lebih dari satu. Berikut ialah contoh kodenya :

```
Buruk
func createFile(name, temp){
	if(temp){
		ioutil.WriteFile(temp)
	}else{
		ioutil.WriteFile(name)
	}
}
```

```
Baik
func createFile(name){
      ioutil.WriteFile(temp)
}

func createTempFile(temp){
      ioutil.WriteFile(temp)
}
```

3. Hindari efek samping

Efek samping disini mengacu pada pengaksesan variable global atau parameter yang di akses oleh *method/*fungsi. Hal ini dapat menjadi fatal karena kesalahan akan sulit untuk di deteksi. Namun di golang sendiri penamaan variable global haruslah diawali dengan huruf besar, sehingga membuat penamaan variable lebih jelas dan mengurangi jumlah error.

```
Variable global yang dapat diakses modul lain
const GlobalVariable = "global"

Variable local yang hanya diakses secara lokal
const localVarible = "local"
```

2.6.3 *Clean Comment*

Pada konsep ini terdapat petunjuk dalam melakukan pemberian komentar yang baik dan efisien, agar komentar yang ditulis dapat memberikan informasi yang tidak tersampaikan oleh kode sumber yang sudah ditulis. Akan tetapi, pemanfaatan *clean comment* berhubungan dengan pemanfaatan *meaningful names,* di mana semakin jelas penamaan dari suatu variabel, *method*/fungsi, maupun *class*/modul semakin sedikit pula komentar yang harus diberikan. Berikut beberapa petunjuk dan aturan pada penulisan *Clean Comment*:

1. Gunakan komentar untuk hal yang meliki kompleksitas logika

   Penulisan kode yang sesuai aturan sebelumnya, akan menghasilkan kode yang mudah dipahami dan jumlah komentar dapat berkurang, Tetapi tidak semua kode harus mengurangi kompleksitas dari kodenya karena Sebagian kode memiliki kompleksitas yang tidak dapat dihindari. Jadi untuk membantu memahmi kode tersebut digunakanlah *Clean Comment* untuk memberikan penjelasan pada bagian kode yang rumit seperti perhitungan atau kalkulasi.

2. Komentar klarifikasi yang informatif

   Sebuah komentar yang baik, dapat menjelaskan informasi dari kode yang ditulis. Berikut adalah contoh dari komentar yang memberikan klarifikasi kodenya.

```
// Mengambil data user dari datbase
Func getUserData();
```

2.6.4 *Clean Formatting*

Pada aturan ini terdapat petunjuk untuk melakukan penulisan kode sumber agar mudah dibaca dengan memanfaatkan format penulisan, seperti *space, tab, indentation* dan penempatan kode sesuai kegunaannya. Penentuan konvensi kode tersebut dapat diambil dari bahasa pemrograman yang digunakan, atau dengan mengikuti kesepakatan yang dibuat oleh tim. Adapun beberapa aturan dan contoh pembuatan *Clean Formatting* sebagai berikut :

1. Konsisten

   Konsistensi terhadap format penulisan penting untuk diperhatikan agar kode yang ditulis seragam dan tidak berantakan. Selain itu konsistensi juga dapat membuat pengembang lain didalam tim mudah untuk memahami kode tersebut.

2. Indentasi

   Pemberian indentasi pada kode program dapat mempermudah pengembang lain untuk mengerti kode yang ditulis, berikut adalah contoh kode :

```
Buruk
func main() {
html, _ := ioutil.ReadFile(tmpDir)
fmt.Println("Raw: \n", html)
fmt.Println("String: \n", string(html))
}

Baik
func main() {
	welcomeData := []byte("Welcome to my website.")

	path := filepath.Join(tmpDir, "/welcome.txt")

	err := ioutil.WriteFile(path, welcomeData, 0777)

	if err != nil {
		fmt.Println(err)
	}

}
```

3. Penempatan *method/*fungsi yang saling berkaitan

   Disarankan untuk memanggil *method*/fungsi yang memilki fungsi saling berkaitan secara berdakatan, hal ini dilakukan untuk mempermudah bila terjadinya kesalahan.

2.6.5 *Clean Object & Data Structure*

Clean Object & Data Strucuture ditujukan untuk membuat struktur data yang memiliki abstraksi sehingga tidak dapat langsung diakses secara publik. Untuk menjadikan sebuah kode memiliki Clean Object & Data Structure ada beberapa hal yang haus diikuti, seperti :

1. Menggunakan private attribute
2. Menggunakan getter dan setter

Berikut adalah contoh kode dari implementasi *Clean Object & Data Structure* :

```
type post struct {
	ID        int    `json:"id"`
	PostName  string `json:"post_name"`
	Slug      string `json:"slug"`
	Content   string `json:"content"`
	UserID    int    `json:"user_id"`
	ViewCount int    `json:"view_count"`
	Createdat string `json:"createdat"`
	Updatedat string `json:"updatedat"`
}

func NewPost(inputData map[string]string) post {
	id, _ := strconv.Atoi(inputData["id"])
	user_id, _ := strconv.Atoi(inputData["user_id"])
	view_count, _ := strconv.Atoi(inputData["view_count"])
	p := post{
		ID:        id,
		PostName:  inputData["username"],
		Slug:      inputData["password"],
		Content:   inputData["password"],
		UserID:    user_id,
		ViewCount: view_count,
		Createdat: inputData["createdat"],
		Updatedat: inputData["updatedat"],
```

```
	}

	return p
}
```

2.6.6 *Clean Error Handling*

Pada bagian ini dijlaskan tentang petunjuk untuk menangani kesalahan (*error)* dengan tepat, penanganan kesalahan ditujukan untuk menampilkan kesalahan dengan mudah, sehingga kesalahan dapat ditinjau dengan baik. Kesalahan yang umum terjadi ialah kesalahan dalam menampilkan *error* yang berulang, Tindakan yang dapat dilakukan ialah dengan membuat notifikasi, ke pengguna dan juga pengembang tentang masalah yang terjadi.

2.6.7 *Clean Classes*

Dalam konsep ini terdapat petunjuk untuk membuat kelas / modul yang lebih bersih dan terorganisir. Berkut adalah beberapa aturan untuk membuat *Clean Classes :*

1. Class Organization

Pada aturan ini dijelaskan bahwa untuk membuat kelas / modul yang baik haruslah mengikuti konvensi dari bahasa pemrograman yang digunakan, dikarenakan bahasa pemrograman yang digunakan adalah golang, maka contoh kode *Clean Classes* ialah sebagai berikut :

```
type PostController struct {
	db *sql.DB
}

func (c *PostController) InitController(db *sql.DB) {
	c.db = db
}
```

2. Single Responsibility Principle

Sebuah kelas/modul yang baik disarankan tidak memiliki ukuran yang terlalu besar dan juga tidak melakukan banyak pekerjaan jenis pekerjaan, dengan menggunakan kelas/modul yang tidak terlalu besar in ikan mempermudah proses modifikasi dan juga pencarian masalah. Kelas/modul yang baik adalah kelas/modul yang hanya memiliki 1 tugas utama saja.

2.7 *Design Pattern*

Design pattern adalah bentuk solusi design software yang dapat digunakan untuk memecahkan masalah yang sering terjadi dalam proses desain aplikasi berorientasi objek, sehingga solusi yang sudah ada dapat digunakan Kembali untuk memecahkan masalah yang serupa, selain itu manfaat lain yang didapat dari implementasi *Design Pattern* ialah seperti berkurangnya *cost* pembangunan aplikasi, berukurang nya waktu yang diperlukan untuk membangun aplikasi, meningkatkan code quality dan maintainability, kode yang ditulis akan menjadi standar umum dikarenakan design pattern yang banyak digunakan secara umum [17].

2.8 *Service Layer Pattern*

Sebuah aplikasi pada umumnya memerlukan banyak interface untuk data yang disimpan dan juga logika untuk melakukan pemrosesan data. Walaupun terdapat perbedaan dari tiap module yang menyimpan dan mengolah logika data tersebut. Sering kali ditemukkan kesamaan ataupun keperluan yang sama di berbagai module nya. Terkadang sebuah interaksi modul dapat memerlukan banyak penyimpanan dan juga pemrosesan logika data yang berebeda-beda. *Service Layer Pattern* membuat sebuah lapisan baru diantara lapisan data dan juga logika bisnis, dengan menambahkan lapisan khusus yang ditujukan untuk mengelola logika dan juga menyimpan data jumlah duplikasi dalam alur proses bisnis dapat berkurang secara signifikan.[18]

2.9 *Coupling and Cohesion*

Coupling dan *Cohesion* adalah istilah yang sering digunakan untuk mengukur keterkaitan dan kegunaan dalam kelas/modul. *Coupling* mengukur

seberapa terkaitnya sebuah modul terhdap modul yang lain, semakin tinggi coupling maka semakin banyak dependensi dari sebuah modul ke modul lainnya hal ini dapat menyebabkan sebuah kode sumber menjadi sulit untuk dilakukan perubahan dan perawatan, umunya *coupling* yang tinggi ingin dihindari dalam proses pengembangan perangkat lunak. [19]

Cohesion mengukur tingkat konektivitas elemen-elemen dalam sebuah modul. Berbeda dengan coupling, semakin tinggi *Cohesion* maka semakin baik, ini mengindikasikan bahwa modul memiliki *single responsibility*. Idealnya pada proses pengembangan perangkat lunak program dapat mencapai tingkat *Low Coupling dan High Cohesion*.

2.10 Performance Testing

Performance Testing dilakukan mengukur dan menialai performa dari sistem, terutama performa dari API yang ada pada sistem. Pengukuran dilakukan dengan cara melakukan request terhadap *Endpoint-endpoint* dari API tertentu secara berulang kali. Dengan begitu akan didapat rata-rata dari response time dari masing-masing *Endpoint.*

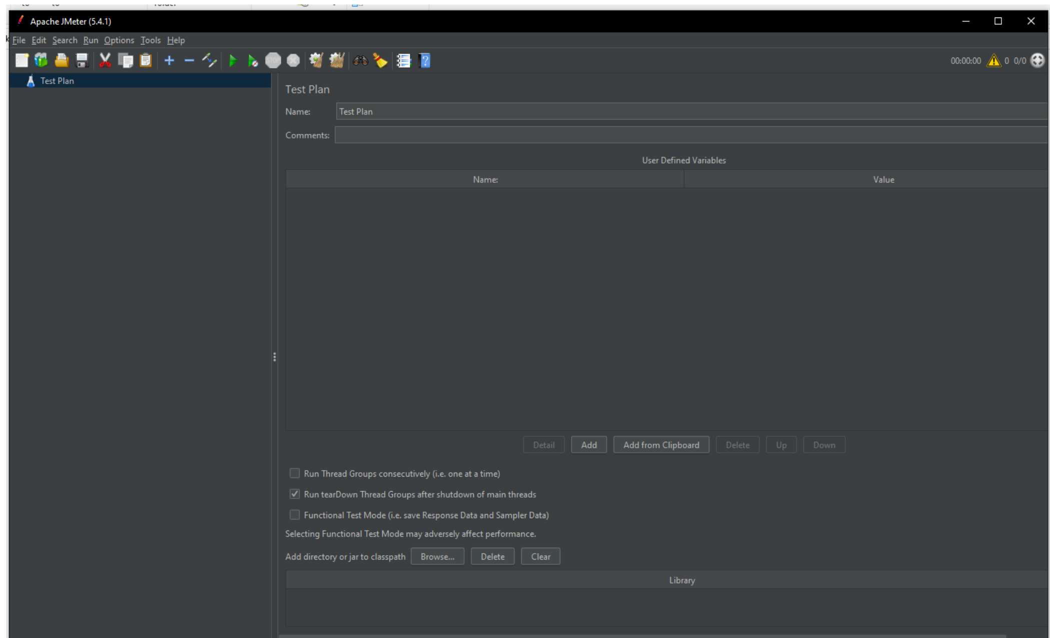
Umumnya *response time* yang baik ada di jangka waktu 0-1 Detik untuk memberikan pengguna sebuah alur penggunaan yang tidak terinrupsi, bila response time sudah melibihi 2 detik ada kemungkinkan alur pengguna akan terganggu, sangat disarankan untuk sebuah sistem dapat memerbikan *response time* dibawah 2 detik [20].

Bila sebuah endopoint memiliki *Response Time* diatas rata-rata atau standar, maka dapat dilakukan optimisasi baik logis maupun data untuk *Enpoint* tersebut. Sehingga dapat dipastikan kepuasan konsumen dalam menggunakan layanan yang ada [21].

2.10.1 Apache JMeter

Apache JMeter adalah aplikasi berbasiskan java dan juga *open source,* Apache Jmeter

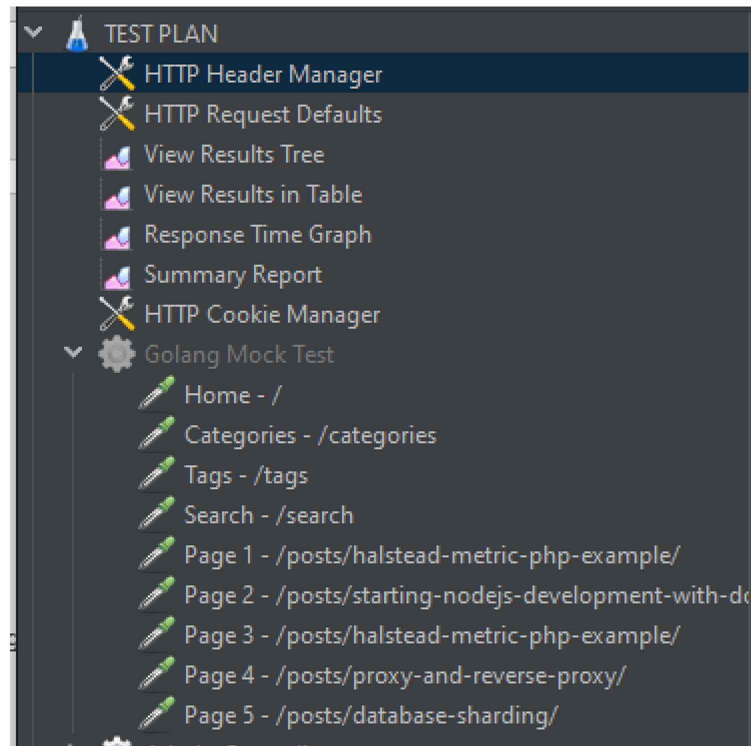
Dibuat untuk melakukan functional testing dan juga performance testing dalam sebuah kondisi. Apache JMeter awalnya dibuat untuk menguji Aplikasi berbasiskan web, namun selain pengujian aplikasi web Apache JMeter juga menawarkan banyak fitur yang dapat digunakan untuk melakukan pengukuran [21].



Gambar 4 Tampilan Awal Apache JMeter

Untuk menjalankan Apache JMeter, pengguna haruslah menajalan *script* .bat atau .sh untuk dapat menjalankan aplikasi GUI nya. Salah satu kelebihan Apache JMeter adalah pengguna tidak memerlukan keahlian pemrograman yang lanjut untuk dapat melakukan *Performance Testing*. Dengan menggunakan GUI yang ada pengguna dapat dengan mudah membuat sebuah *Performance Testing*.

Pengguna dapat membuat sebuah Test Plan yang berisikan bermacam-macam thread group, dari masing-masing thread group ini pengguna dapat menentukan berapa banyak *Thread* dan jumlah iterasi yang akan digunakan.



Gambar 5 Contoh Tampilan *Test Plan*

Di dalam Thread Group, pengguna dapat membuat HTTP Request untuk melakukan Request terhadap sistem yang akan diujikan, hasil dari pengujian dapat dilihat dalam berbagai macam output format diantara nya :

1. Result Tree
2. Summary Report
3. Aggregate Report
4. Aggregate Graph
5. Assertion Results
6. Generate Summary results
7. Graph Results
8. Result Table

Hasil test juga dapat dibuat menjadi bentuk CSV (*comma-separated values*) yang dapat diolah lebih lanjut menjadi hasil analisis yang lebih mendalam [22].