

**İSTANBUL TEKNİK ÜNİVERSİTESİ  
BİLGİSAYAR VE BİLİŞİM FAKÜLTESİ**

**NESNEYE DAYALI PROGRAMLAMADA  
KUSURLU KODLARIN BELİRLENMESİ**

**Bitirme Ödevi**

**Murat Sekin  
040040243**

**Bölüm : Bilgisayar Mühendisliği  
Anabilim Dalı: Bilgisayar Bilimleri**

**Danışman : Doç.Dr. Feza BUZLUCA**

**Haziran 2016**

**İSTANBUL TEKNİK ÜNİVERSİTESİ  
BİLGİSAYAR VE BİLİŞİM FAKÜLTESİ**

**NESNEYE DAYALI PROGRAMLAMADA  
KUSURLU KODLARIN BELİRLENMESİ**

**Bitirme Ödevi**

**Murat Sekin  
040040243**

**Bölüm : Bilgisayar Mühendisliği  
Anabilim Dalı : Bilgisayar Bilimleri**

**Danışman : Doç.Dr. Feza BUZLUCA**

**Haziran 2016**

## **Özgünlük Bildirisi**

1. Bu çalışmada, başka kaynaklardan yapılan tüm alıntılar, ilgili kaynaklar referans gösterilerek açıkça belirtildiğini,
2. Alıntılar dışındaki bölümlerin, özellikle projenin ana konusunu oluşturan teorik çalışmaların ve yazılım/donanımın benim tarafımdan yapıldığını bildiririm.

İstanbul, 03.06.2016

Murat Sekin

# NESNEYE DAYALI PROGRAMLAMADA KUSURLU KODLARIN BELİRLLENMESİ

## ( ÖZET )

Kaynak kodlaron derlenmesini ve çalışmasını engellemeyen ancak daha derin sorunların işaretçisi olan problemlere kusurlu kod denir. Kusurlu kodlar yazılımın okunmasını, anlaşılmasını, güncellenmesini zorlaştırır. Gelecekte yeni hataların çıkmasına neden olur. Yazılım kalitesini düşürür. Para ve zaman kaybına neden olabilir. Bu nedenle kusurlu kodların belirlenip bu kusurların giderilmesi gerekir. Kaynak kodların analiz edilerek kusurların belirlenmesi işleme statik kod analizi adı verilir. Analiz işleminden önce kodun parse edilmesi gerekir. Kaynak kodları belli bir formata sahip olduğundan ve programlama dilinin grameri de mevcut olduğundan parser generator kullanabilir.. Parse generator olarak ANTLR tercih edildi. ANTLR ile kendi parserımızı oluşturmuş olduk. Parser sonucunu anlamdırarak kusurlu kodlar bulunur.

# **DETECTING BAD CODES IN OBJECT-ORIENTED PROGRAMS**

## **( SUMMARY )**

Bad codes don't prevent compiling and execution of source codes. They are indications of deeper problems. They are generally caused by violation of design guidelines and coding standards. It is hard to read, understand and update source codes with bad codes. This decreases software quality and makes new bugs to arise in the future. It can cause loss of time and money. Therefore, defects must be identified and to be fixed. Analyzing the source code without executing is called static code analysis. In the first step the source code must be parsed. As source codes are structured text, a parser generator can be used. In this project ANTLR parser generator is being used. We create our own parser with parser generator. As the output of parser is a parse tree, we have to traverse the tree data structure to find bad codes.

# İÇİNDEKİLER

1. GİRİŞ .....	1
2. PROJENİN TANIMI VE PLANI .....	2
2.1. Projenin Tanımı .....	2
2.2. Proje Planı .....	2
2.2.1. Projenin Amacı .....	2
2.2.2. Projenin Kapsamı .....	3
2.2.3. Zamanlama .....	3
3. KURAMSAL BİLGİLER .....	5
3.1. Kusurlu Kod .....	5
3.2. Statik Kod Analizi .....	5
3.3. Yazılım Metrikleri .....	5
3.4. Parsing .....	5
3.5. ANTLR .....	6
4. ANALİZ VE MODELLEME .....	9
4.1. Analiz .....	9
4.2. Modelleme .....	10
5. TASARIM, GERÇEKLEME VE TEST .....	14
5.1. Tasarım .....	14
5.2 Gerçekleme .....	16
5.2.1. Kullanılan Teknolojiler .....	16
5.2.2. Kaynak Kodun Parse Edilmesi .....	16
5.2.3. Tespit Edilen Kusurlar .....	17
5.2.3.1. If içerisinde atama operatörü kullanılması .....	17
5.2.3.2. Boş catch bloğu .....	17
5.2.3.3. Goto deyiminin kullanımı .....	17
5.2.3.4. Fonksiyonun uzun olması .....	17
5.2.3.5. İç içe yuvalanmış çok sayıda if bloğu .....	17
5.2.3.6. Fonksiyonların çok sayıda parametreye sahip olması .....	17
5.2.3.7. Lack of Cohesion of Methods (LCOM) .....	17
5.2.3.8. Coupling Between Objects (CBO) .....	18
5.3. Test .....	19
6. DENEYSEL SONUÇLAR .....	20
7. SONUÇ ve ÖNERİLER .....	21
8. KAYNAKLAR .....	22
EKLER .....	23
EK 1. Kusurlu Kod Listesi .....	23

## ŞEKİLLER

Şekil 2.1: Kötü kod örnekleri .....	3
Şekil 2.2: Zaman Çizelgesi.....	3
Şekil 2.3: Gantt diyagramı.....	4
Şekil 3.1: ANTLR C++ grammarine ait bir bölüm .....	6
Şekil 3.2: “Hello World” programı .....	6
Şekil 3.3: ANTLR Lexer sonucu .....	7
Şekil 3.4: ANTLR Lexer kayıtlarının açıklaması.....	7
Şekil 3.5: ANTLR parse tree .....	8
Şekil 4.1: İş akış diyagramı .....	10
Şekil 4.2: Proje içerisindeki sınıflar ve arayüzler .....	11
Şekil 4.3: UML Diyagramı 1 (Global namespace).....	12
Şekil 4.4: UML Diyagramı 2 (Global namespace).....	12
Şekil 4.5: UML Diyagramı 3 (CodeAnalysis.Checks namespace) .....	13
Şekil 4.6: UML Diyagramı 4 (CodeAnalysis.Checks.Metrics namespace) .....	13
Şekil 5.1: Ana ekran .....	14
Şekil 5.2: Ayarlar ekranı.....	15
Şekil 5.3: Hakkında ekranı .....	16
Şekil 5.4: Kaynak kodun parse edilmesi .....	16
Şekil 5.5: LCOM2 metriğinin şekil olarak gösterimi .....	18
Şekil 5.6: CBO metriğinin şekil olarak gösterimi .....	18

# TABLolar

<b>Tablo 4.1:</b> C++ versiyonları .....	9
<b>Tablo 5.1:</b> CBO eşik değerleri .....	19



# 1. GİRİŞ

Yazılım kalitesini arttırmak için statik kod analizi günümüzde sıkça kullanılmaktadır. Özellikle kritik uygulamalarda bu tür programların kullanılması daha bir önem kazanmıştır. Çünkü derleyiciler sadece dilin kurallarına uymayan hataları bulabilir. Statik kod analizi ile kodlar çalıştırılmadan analiz edilip kusurlar bulunur. Bu tür kusurlar programın derlenmesini çalışmasını etkilemese de ilerde sorun çıkarma olasılığı yüksektir. Bu proje kapsamında da kusurlu kodlar hakkında araştırma yapıp bir statik kod analiz programı geliştirilecektir. Açık kaynak kodlu veya ticari olmak üzere çok sayıda statik analiz programı bulmak mümkündür[19].

Projeye en başında kusurlu kodlar hakkında araştırma yapılarak başlanmıştır. Bu araştırmanın sonucuna EK 1 Kusurlu Kod Listesi'nden ulaşılabilir.

Kaynak kodun parse edilebilmesi için ANTLR (ANother Tool for Language Recognition) isimli parser generator kullanılmasına karar verilmiştir. ANTLR hakkında detaylı bilgiye kurumsal bilgiler bölümünden ulaşılabilir.

Gerçeklenen programın son haline 5. Tasarım, Gerçekleme ve Test bölümünden ulaşılabilir.

Rapor içerisinde kusurlu kod ve kötü kod aynı anlamda kullanılmıştır.

## 2. PROJENİN TANIMI VE PLANI

### 2.1. Projenin Tanımı

Nesneye dayalı programlamada kusurlu kodların belirlenmesi projesi, kaynak kodların analiz edilerek özelliklerinin elde edilmesi ve problemlerinin bulunmasıdır.

Kullanıcının seçtiği dosyalar ilk önce parse edilip daha sonra anlamlandırılarak problemler tespit edilecek ve kullanıcıya sonuçlar gösterilecektir.

### 2.2. Proje Planı

#### 2.2.1. Projenin Amacı

Nesneye yönelik programlama dili kullanılarak yazılmış kaynak kodlarda, kötü kodları tespit edecek bir program geliştirilecektir. Yazılım kalitesinin artırılması ve iyi kod yazımının özendirilmesi amaçlanmaktadır.

Yazılım kalitesini artırmak için çeşitli standartlar geliştirilmiştir. Bugün iyi olarak kabul edilen kodlama teknikleri de mevcuttur. Bunlara uyulmaması kötü kod yazılmasına neden olabilir. Kötü yazılmış bir kodun okunması, anlaşılması ve bakımı zordur. Programların performansını etkileyebilir ve ilerde hata çıkmasına neden olabilir. Kodların derlenmesini ve çalışmasını engellemez. Şekil 2.1’de C++ programlama dilinde bazı kötü kod örnekleri gösterilmiştir.

Kötü kodların nedenleri:

- Analiz ve tasarımın yeterli ve doğru olmaması
- Yazım kurallarına ve isimlendirme kurallarına uyulmaması
- Tutarsızlık
- Biçime ve formata dikkat edilmemesi
- Tekrar eden kod parçaları
- Yorum olmaması

```

// goto kullanmak. program akışını takip etmeyi zorlaştırır
goto label;
...
label: statement;

// büyük boyutlu vektör fonksiyondan geri döndürmek
std::vector<std::string> BuildLargeVector();
...
std::vector<std::string> v = BuildLargeVector();

// magic number kullanmak
for(int i=0; i < 10; i++)
{
    ...
}

```

Şekil 2.1: Kötü kod örnekleri

## 2.2.2. Projenin Kapsamı

Hedef dil olarak C++ programlama dili seçilmiştir. Bu programlama dili kullanılarak yazılmış programlardaki kötü kodlar bulunacak ve kullanıcıya gösterilecek.

Masaüstü uygulaması olacak.

C# programlama dili ile geliştirilecek.

.NET platformu üzerinde çalışacak.

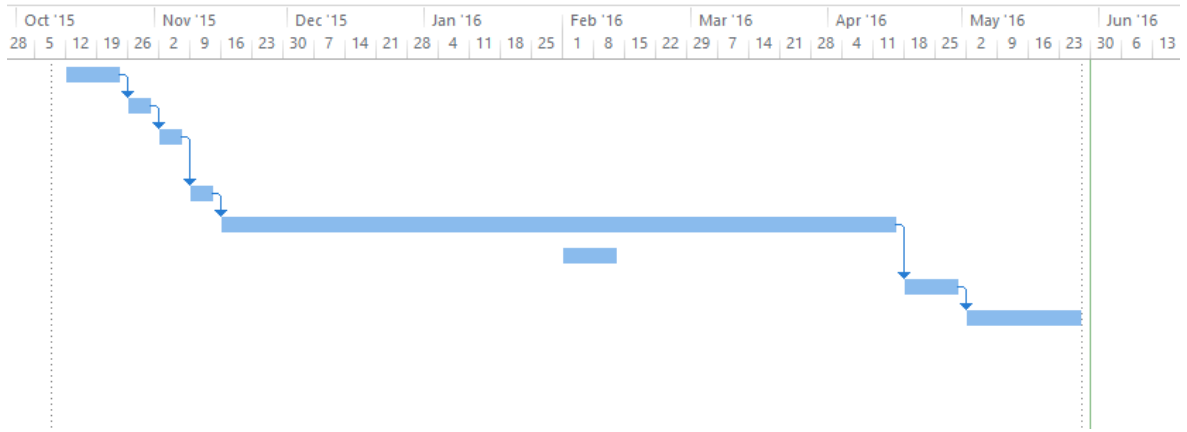
C++ dilinin parse edilmesi için 3. parti bir yazılım kullanılacak.

## 2.2.3. Zamanlama

Projenin iki önemli aşaması bulunmaktadır. İlk olarak kaynak kodların parse edilmesi gerekmektedir. C++'ın karmaşıklığı nedeniyle bu iş için 3. parti yazılım kullanılacak. İkinci aşama ise parserin çıktılarının anlamlandırılmasıdır.

Task Mode ▾	Task Name ▾	Duration ▾	Start ▾	Finish ▾	Predecessors ▾
	C++ öğrenilmesi	10 days	12.10.2015	23.10.2015	
	Parser araştırılması	5 days	26.10.2015	30.10.2015	1
	Kötü kod örneklerinin araştırılması	5 days	2.11.2015	6.11.2015	2
	Tasarım	5 days	9.11.2015	13.11.2015	3
	Kodlama	110 days	16.11.2015	15.4.2016	4
	Ara rapor yazımı	10 days	1.2.2016	12.2.2016	
	Test	10 days	18.4.2016	29.4.2016	5
	Rapor yazımı	20 days	2.5.2016	27.5.2016	7

Şekil 2.2: Zaman Çizelgesi



**Şekil 2.3:** Gannt diyagramı

## 3. KURAMSAL BİLGİLER

### 3.1. Kusurlu Kod

Kaynak kodun derlenmesini ve doğru çalışmasını engellemeyen ancak sistemde daha derin problemlerin işaretçisi olan kod parçalarıdır.[3] Kodun geliştirilmesini, bakımını, anlaşılmasını zorlaştırır. Gelecekte yapılacak değişikliklerde hata çıkması olasılığını arttıran bir etkidir. Kodun kullanılabilirliğini düşürür. Örneğin fonksiyonların sadece tek bir işlem yapması ve mümkün olduğunca kısa olması istenir.

Maliyet (para ve zaman) artışına neden olabilir. Günümüze kadar geliştirilmiş yazılım standartlarına uyulmaması kusurlu kodların oluşmasının en önemli nedenlerinden birisidir. [13] Yazılım kalitesini arttırmak için kusurlu kodların temizlenmesi gerekir.

“Code smell”, “bad practices” kod kusurlarıdır.

Tam listeye raporun sonundaki EK 1 bölümünde bulunmaktadır.

### 3.2. Statik Kod Analizi

Kaynak kodun derlenip çalıştırılmadan incelenmesi işlemine denir.[4] Kusurlu kodlar bu süreçte elde edilir. Statik kod analizi gözden geçirme ile yapılabileceği gibi çeşitli araçlar kullanılarak otomatik olarak yapılabilir.[8]

Statik kod analizinde incelenen konular[8]

- Hata ve Güvenlik Açıkları
- Kaynak Kod Metrikleri
- Mimari Analiz
- Kodlama Standartları
- Tersine Mühendislik

Günümüzde farklı programlama dilleri ve farklı işlemler için birçok statik kod analizi aracı mevcuttur.[19] OCLint ve Cppcheck bu programlardan sadece iki tanesidir. [5],[6]

### 3.3. Yazılım Metrikleri

Yazılım metriği bir yazılımın niteliğinin veya teknik özelliklerinin ölçümüdür.[18] Örneğin number of lines of code metriği bir yazılımdaki kod satırlarını sayar.

### 3.4. Parsing

Bir metni, söz dizimi kurallarına göre daha küçük parçalara ayırma işlemine denir.

### 3.5. ANTLR

ANTLR (ANother Tool for Language Recognition) bir parser generatordur. Parser generatorlar belli bir formattaki metinlere ait şablonları kullanarak parser oluştururlar. Şablonlara grammar ismi verilir.

Projede <https://github.com/antlr/grammars-v4/blob/master/cpp/CPP14.g4> adresindeki grammar kullanılmıştır. Şekil 3.1’de bu grammare ait başlangıç bölümü görünmektedir.

```
grammar CPP14;

/*Basic concepts*/
translationunit
:
    declarationseq? EOF
;

/*Expressions*/
primaryexpression
:
    literal
    | This
    | '(' expression ')'
    | idexpression
    | lambdaexpression
;

idexpression
:
    unqualifiedid
    | qualifiedid
;

...
```

**Şekil 3.1:** ANTLR C++ grammarine ait bir bölüm

ANTLR’nin çalışmasını daha iyi anlayabilmek için Şekil 3.2’deki örnek “Hello World” programına ait parser çıktıları verilmiştir. Parse işlemi sonucunda parse tree oluşmaktadır.

```
#include <iostream>

int main()
{
    std::cout << "Hello World!";
    return 0;
}
```

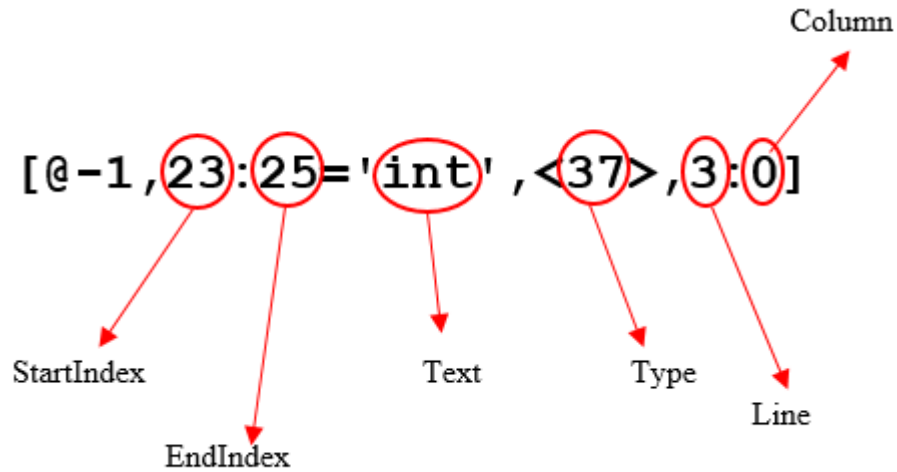
**Şekil 3.2:** “Hello World” programı

```

[@-1,23:25='int',<37>,3:0]
[@-1,27:30='main',<124>,3:4]
[@-1,31:31='(',<77>,3:8]
[@-1,32:32=')',<78>,3:9]
[@-1,35:35='{',<81>,4:0]
[@-1,40:42='std',<124>,5:2]
[@-1,43:44='::',<119>,5:5]
[@-1,45:48='cout',<124>,5:7]
[@-1,50:51='<<',<104>,5:12]
[@-1,53:66='"Hello World! "',<133>,5:15]
[@-1,67:67=';',<120>,5:29]
[@-1,72:77='return',<51>,6:2]
[@-1,79:79='0',<125>,6:9]
[@-1,80:80=';',<120>,6:10]
[@-1,83:83='}',<82>,7:0]
[@-1,84:83='<EOF>',<-1>,7:1]

```

Şekil 3.3: ANTLR Lexer sonucu



Şekil 3.4: ANTLR Lexer kayıtlarının açıklaması

**StartIndex:** Dosyanın başlangıcından itibaren karakter sayısı

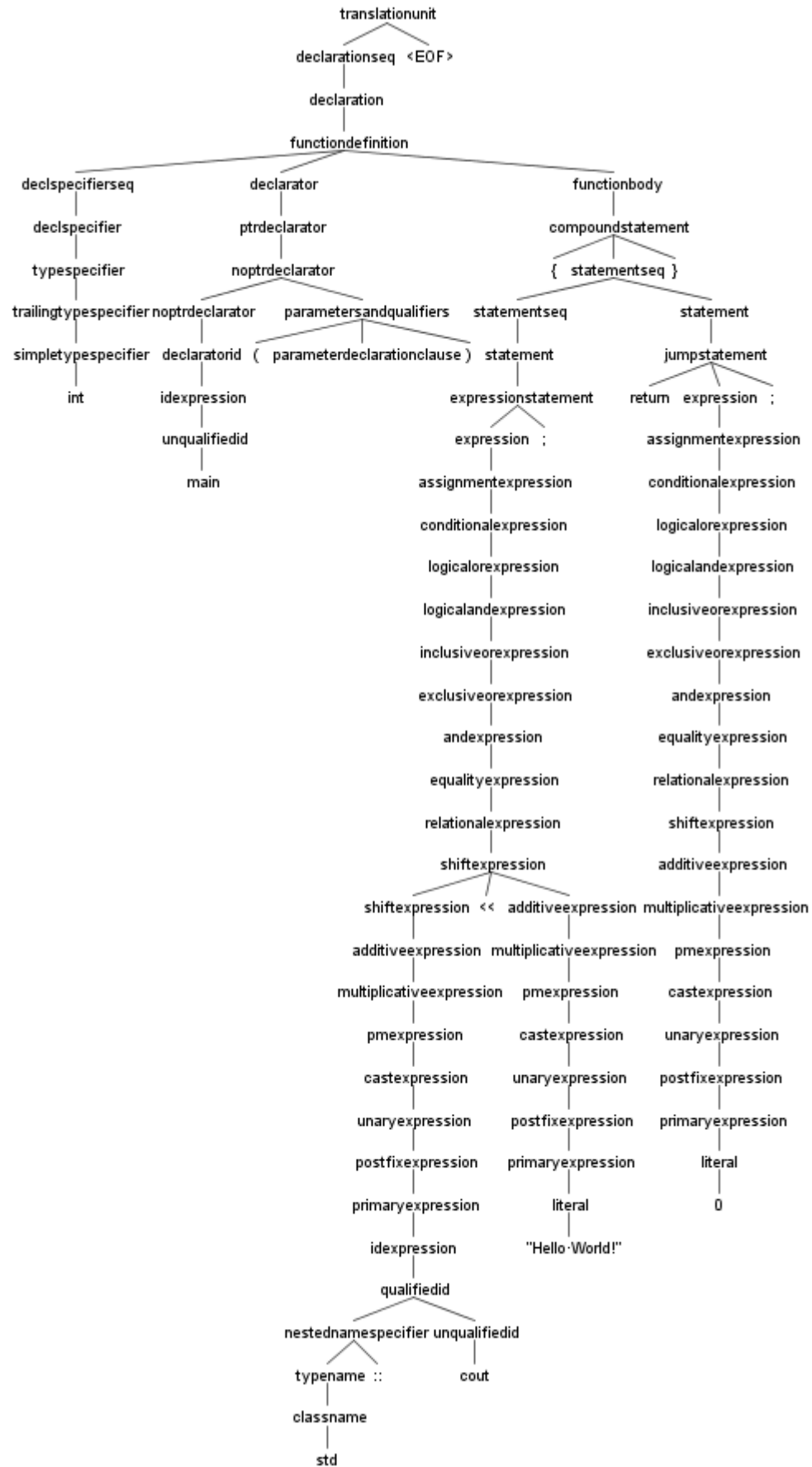
**StopIndex:** **StartIndex** + **Text**'in uzunluğu

**Text:** Kod içerisindeki kelime

**Type:** StringLiteral, Identifier, Int, If, vs. gibi tipler

**Line:** Satır numarası

**Column:** Sütun numarası



Şekil 3.5: ANTLR parse tree



## 4. ANALİZ VE MODELLEME

### 4.1. Analiz

İlk öncelikle kusurlu kodlar ile ilgili araştırma yapılmış ve EK 1 Kusurlu Kod Listesi oluşturulmuştur.

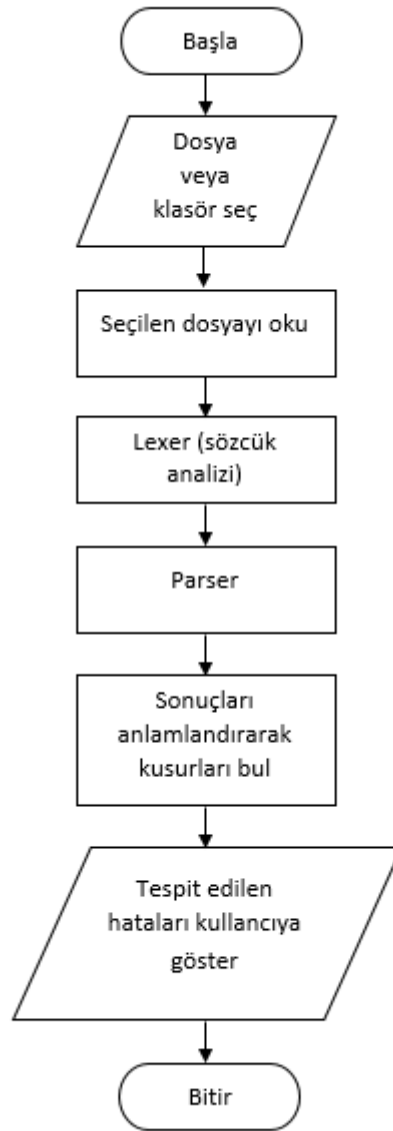
İkinci olarak programın şekil 4.1'deki genel iş akışı belirlenmiştir. Böylece arayüzün taslağı oluşturuldu.

Hedef dil olarak C++ seçildi, bölümde öğretilmesi ve popüler bir dil olması nedeniyle. C++'ın birkaç farklı versiyonu mevcut.

**Tablo 4.1:** C++ versiyonları

versiyon	açıklama
C++98	C++ 1998/2003 Standard
C++11	C++ 2011 Standard
C++14	C++ 2014 Standard
C++17	-

Proje boyunca C++14'ün standardı takip edilmiştir.



**Şekil 4.1:** İş akış diyagramı

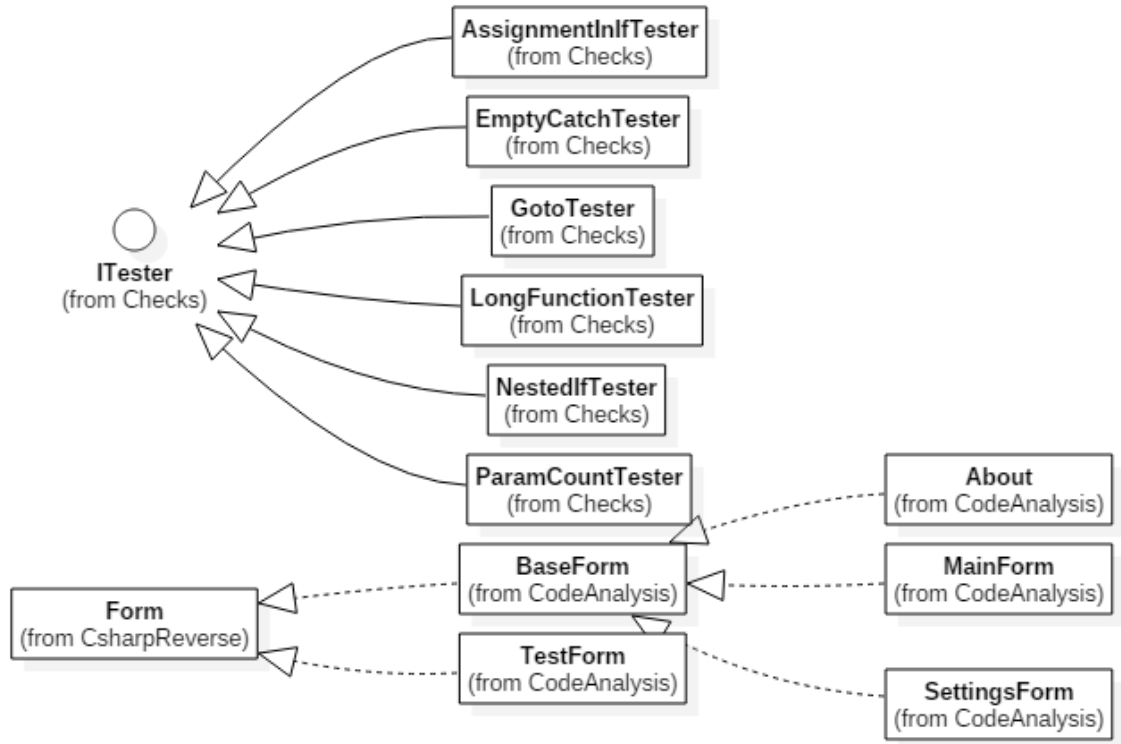
Şekil 4.1’de görüldüğü üzere bir parser gerekmektedir. Bunun için 3 farklı yol bulunmakta:

1. Yeni bir parserı sıfırdan yazmak
2. Açık kaynak kodlu bir derleyicinin front-endini (lexer, parser, preprocess) kullanmak  
Örneğin; clang[20].
3. ANTLR gibi parser generator kullanmak

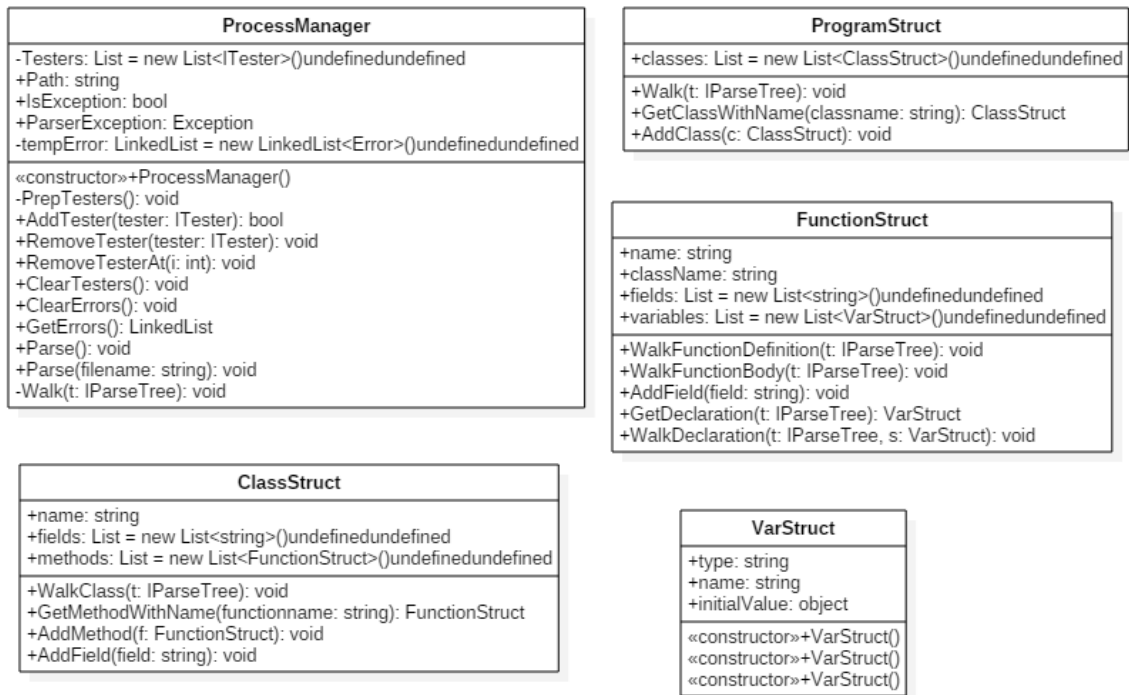
C++ dilinin karmaşık olması ve derleyici tasarımı dersi almadığım için birinci yolu kullanmaktan vazgeçtim. Üçüncü seçenek ikinciye göre daha kolay olduğu için bu yolu tercih ettim. 3.4. bölümde ANTLR detaylı olarak açıklanmıştır.

## 4.2. Modelleme

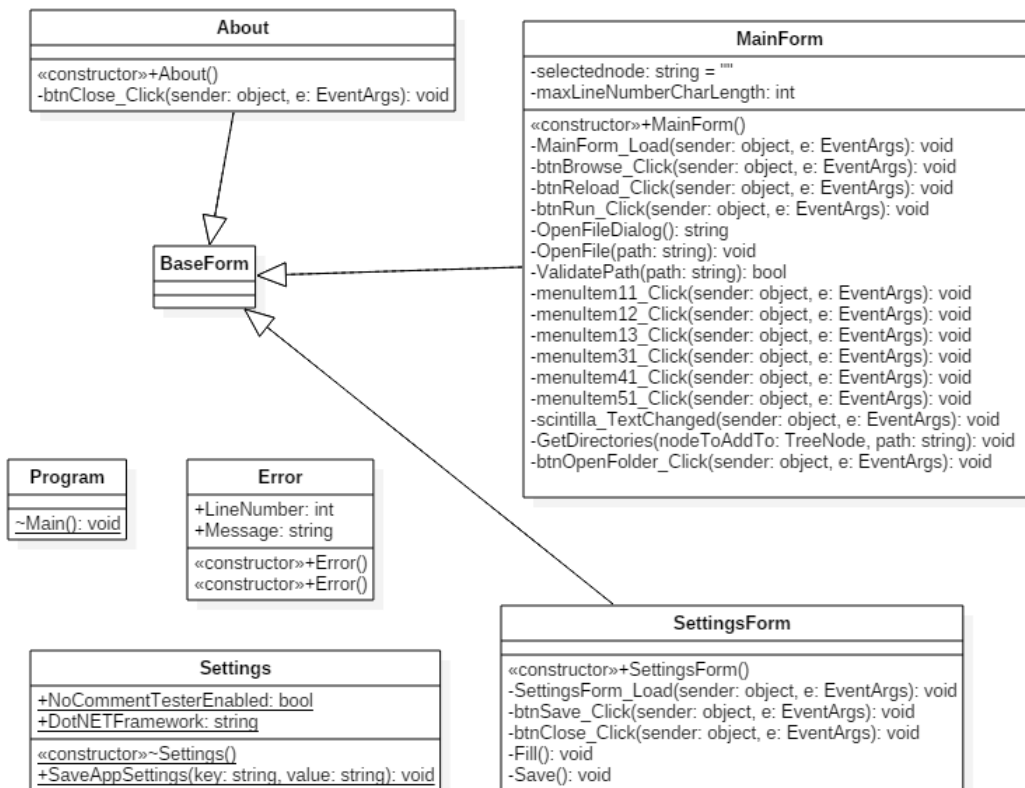
Yazılım modeline ait UML diyagramları aşağıda verilmiştir.



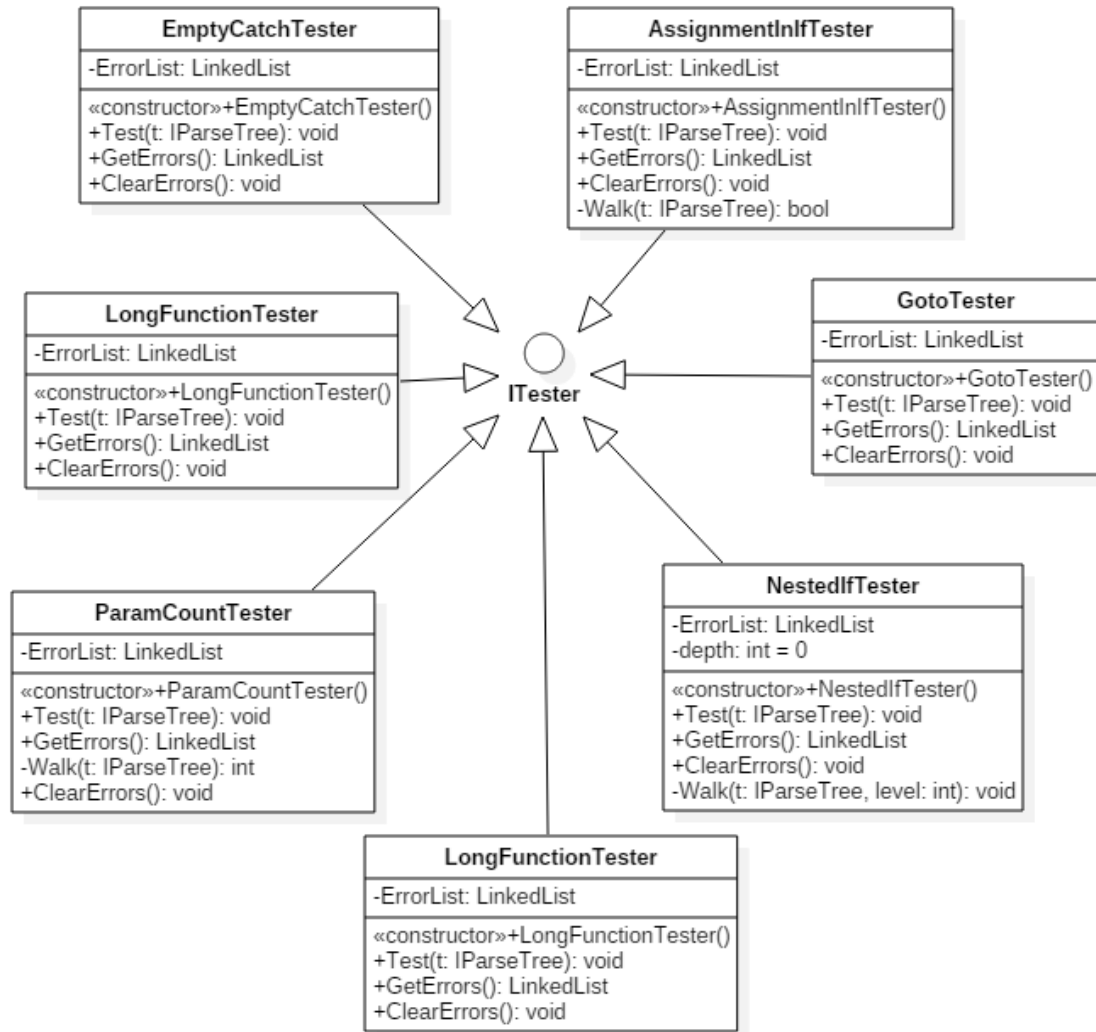
Şekil 4.2: Proje içerisindeki sınıflar ve arayüzler



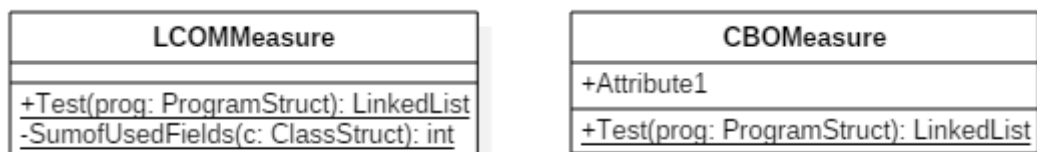
Şekil 4.3: UML Diyagramı 1 (Global namespace)



Şekil 4.4: UML Diyagramı 2 (Global namespace)



Şekil 4.5: UML Diyagramı 3 (CodeAnalysis.Checks namespace)

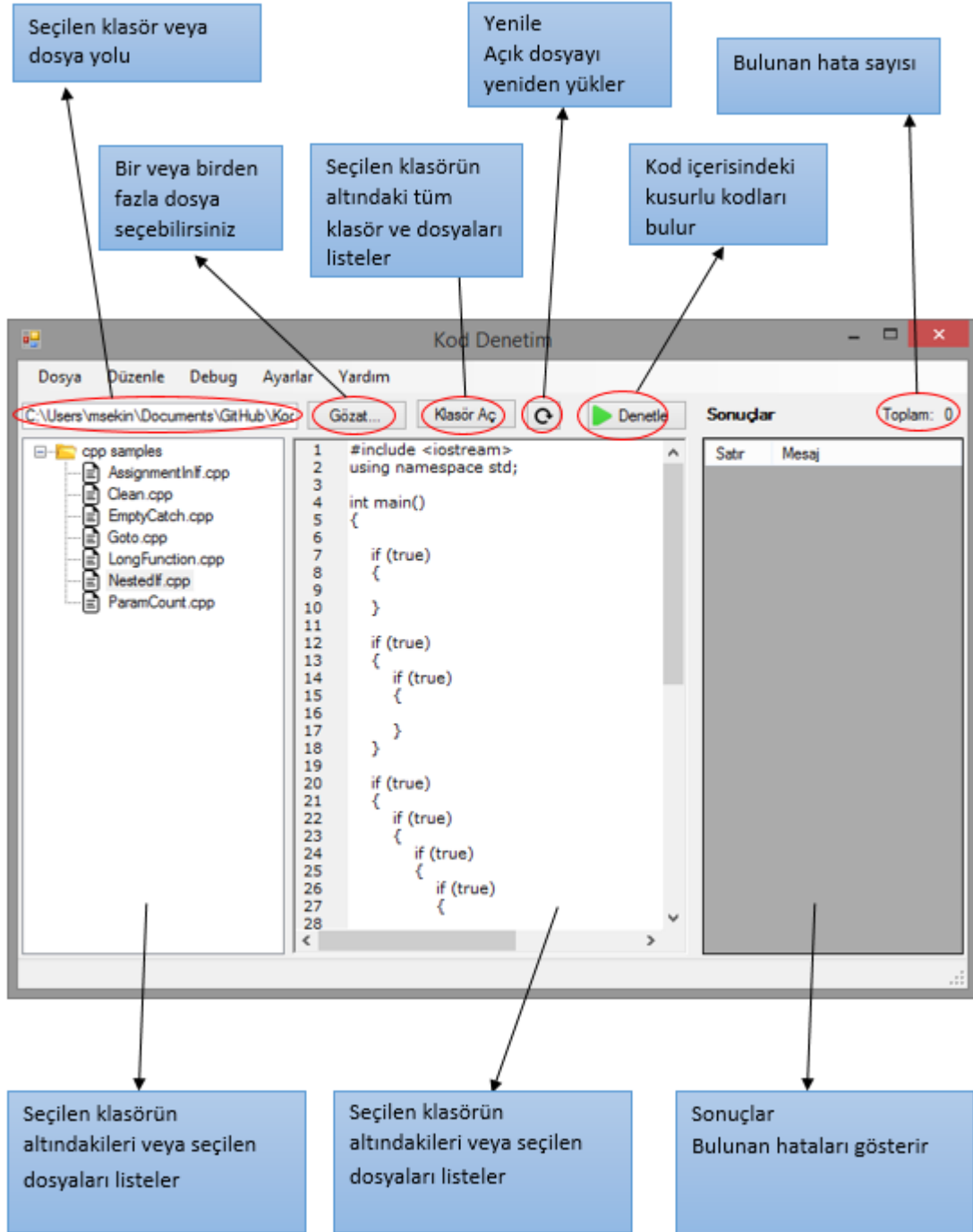


Şekil 4.6: UML Diyagramı 4 (CodeAnalysis.Checks.Metrics namespace)

## 5. TASARIM, GERÇEKLEME VE TEST

### 5.1. Tasarım

Ekran görüntüleri açıklamalarıyla birlikte aşağıda verilmiştir.

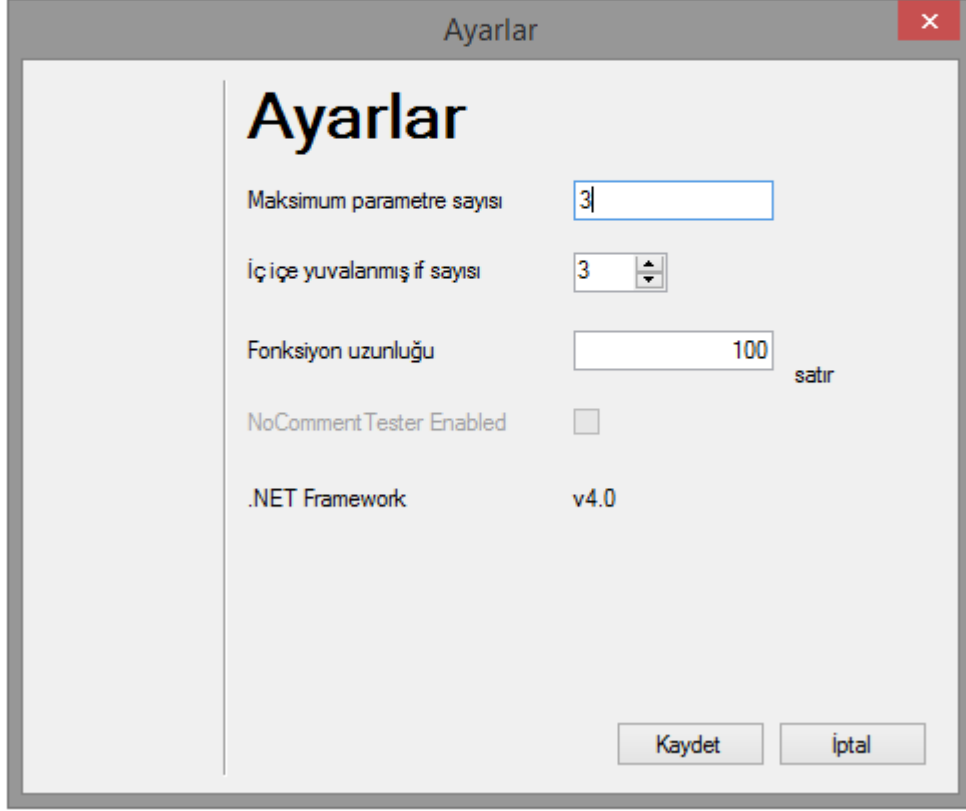


Şekil 5.1 : Ana ekran

Ayarlar ekranı.

Kusurlu kodların eşik değerlerini kaydetmek için kullanılır.

Örneğin parametre sayısı 3'ten fazla olan fonksiyonlar için uyarı verir.



Ayarlar

## Ayarlar

Maksimum parametre sayısı

İç içe yuvalanmış if sayısı

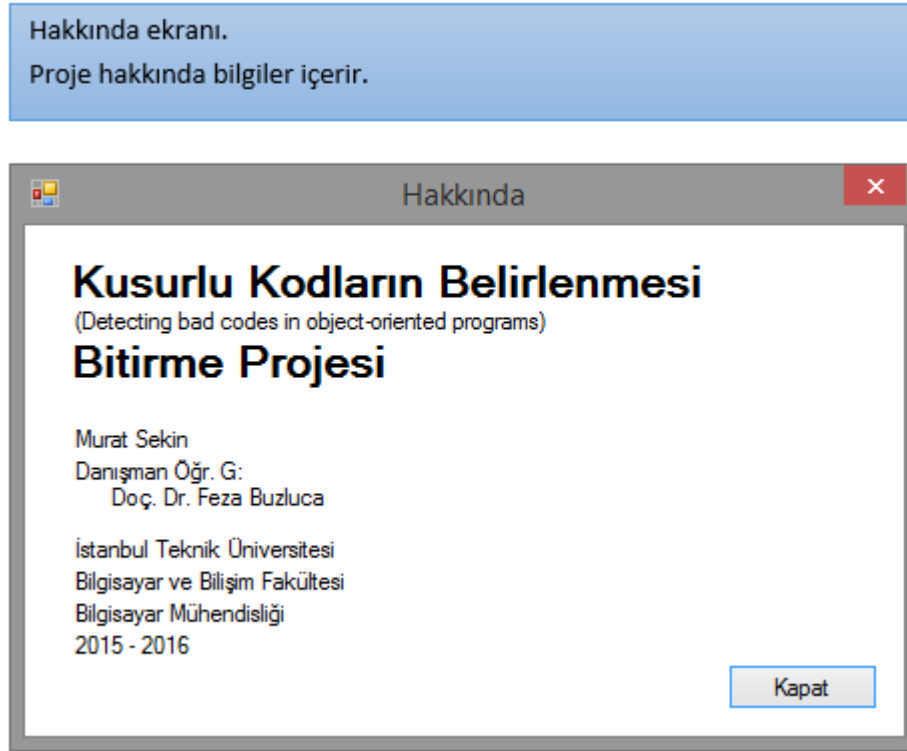
Fonksiyon uzunluğu  satır

NoCommentTester Enabled ☐

.NET Framework v4.0

Kaydet İptal

Şekil 5.2: Ayarlar ekranı



Şekil 5.3: Hakkında ekranı

## 5.2 Gerçekleme

### 5.2.1. Kullanılan Teknolojiler

Geliştirme Microsoft Visual Studio 2013 geliştirme ortamı üzerinde yapılmıştır. Uygulama C# programlama dili kullanılarak yazılmıştır. Uygulamanın çalışması için .NET Framework 4.0'a ihtiyaç vardır.

### 5.2.2. Kaynak Kodun Parse Edilmesi

Kaynak kodu parse etmek için kullanılan kodun bir kısmı aşağıda verilmiştir. Bu işlemin sonucunda bir ağaç yapısı (parse tree) döndürmektedir.

```
using (FileStream filestream = new FileStream(Path, FileMode.Open))
{
    AntlrInputStream inputstream = new AntlrInputStream(filestream);
    CPP14Lexer lexer = new CPP14Lexer(inputstream);
    CommonTokenStream tokens = new CommonTokenStream(lexer);
    CPP14Parser parser = new CPP14Parser(tokens);
    CPP14Parser.TranslationUnitContext translationunit =
    parser.translationunit();

    Walk(translationunit);
}
```

Şekil 5.4: Kaynak kodun parse edilmesi



### 5.2.3. Tespit Edilen Kusurlar

Kusurlar 5.2.2'deki parse tree dolaşarak veya bu ağaç yeni bir yapıya aktararak bulunmaktadır.

#### 5.2.3.1. If içerisinde atama operatörü kullanılması

`==` (eşitlik) operatörü yerine `=` (atama) operatörünün kullanılması durumudur. Program istenilenden farklı bir işlevi yerine getirir.

#### 5.2.3.2. Boş catch bloğu

Exceptionlar yutulmamalı. Eğer istisnai durum yönetilmeyecekse yakalanmamalı.

#### 5.2.3.3. Goto deyiminin kullanımı

"goto" deyiminin kullanımı kodun okunmasını zorlaştırır, oluşan dallanmalardan dolayı karmaşıklığı artırır.

#### 5.2.3.4. Fonksiyonun uzun olması

Fonksiyonlar tek bir iş yapmalı. Uzun olması okunması ve anlaşılmasını zorlaştırır.

#### 5.2.3.5. İç içe yuvalanmış çok sayıda if bloğu

Çok sayıda farklı yol kodun kontrolünü zorlaştırır, karmaşıklığını artırır.

#### 5.2.3.6. Fonksiyonların çok sayıda parametreye sahip olması

Fonksiyonların birden fazla iş yaptığının işaretçisi olabilir.

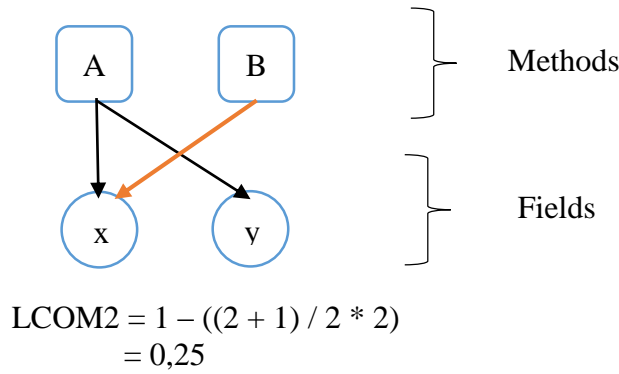
#### 5.2.3.7. Lack of Cohesion of Methods (LCOM)

Cohesion bir sınıf içerisindeki niteliklerin ne kadar birbiriyle ilişkili olduğunu belirtir[7]. İstenilen, sınıfların yüksek cohesion'a (high cohesion) sahip olmasıdır. LCOM'ın birçok türü vardır: LCOM1, LCOM2, LCOM3, LCOM4, LCOM5. Hesaplamalarda LCOM2 kullanıldı[10].

m	methodların sayısı
a	niteliklerin sayısı
mA	bir niteliğe erişen methodların sayısı
sum(mA)	mA'ların toplamı

$$LCOM2 = 1 - \text{sum}(mA)/(m*a) \text{ [10]}$$

LCOM2 0 (high cohesion) ve 1 (no cohesion) değerleri arasında değişir.



Şekil 5.5: LCOM2 metriğinin şekil olarak gösterimi

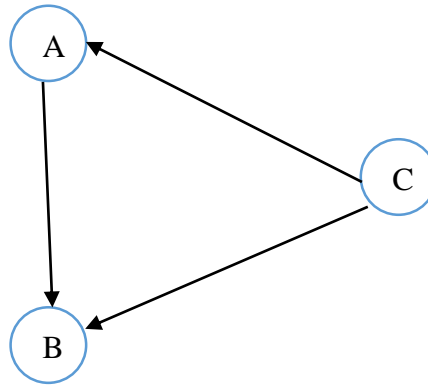
LCOM2 için eşik değeri 0,8 alınmıştır. Ticari bir uygulama olan NDepend programının eşik değeri kullanılmıştır[11]. Eğer istenirse Ayarlar ekranından eşik değeri ayarlanabilir.

### 5.2.3.8. Coupling Between Objects (CBO)

Coupling bir sınıfın diğer sınıflara ne kadar bağlı olduğunu belirtir[7]. Arzu edilen düşük coupling (low coupling) tir. Yani sınıfların birbirlerine bağlarının az olmasıdır. Böylece kodun kullanılabilirliği artar.

CBO = Bir sınıfın bağlı olduğu sınıflar [12].

Eğer A sınıfı, B sınıfına ait bir metodu çağırıyor veya B sınıfına ait bir değişkene erişiyor veya B tipinde bir nesneyi metodlarına parametre olarak geçiriyorsa veya methodlarından B sınıfı tipinde nesne döndürüyorsa bu durumda A, B'ye bağlıdır.



Şekil 5.6: CBO metriğinin şekil olarak gösterimi

$$A_{cbo} = 1$$

$$B_{cbo} = 0$$

$$C_{cbo} = 2$$

**Tablo 5.1:** CBO eşik değerleri [9]

<b>Geliştirici</b>	<b>Eşik Değeri</b>
Rosenbar, NASA	5
SD-Metrics	0-31
Together Soft	30
Objectteering Enterprise Edition	1-4

Farklı geliştiricilerin CBO için kullandıkları eşik değerler geliştirdiğimiz programda referans alındı. Ayarlar ekranında ayarlanabiliyor. Eşik değeri aşması durumunda kullanıcıya uyarı veriliyor.

### 5.3. Test

Her bir kusurlu kod teker teker test edilmiştir. Testler basit ve karmaşık C++ kodlarından oluşmaktadır. Yine ayrıca her bir fonksiyonel işlem test edilmiştir.

## 6. DENEYSEL SONUÇLAR

Testler esnasında karşılaşılan hatalar giderilmiştir.

## 7. SONUÇ ve ÖNERİLER

Geliştirme ve test aşamasında iyileştirilebilir üç durum tespit edildi.

Geliştirilmiş olan program şuan için sadece tek bir dosya üzerinde çalışabiliyor. Aynı anda çoklu dosyaları denetleyemiyor. Örneğin bir sınıfın declaration'ı ve definition'ı aynı dosyada yer almalı.

ANTLR'nin oluşturduğu parse tree'nin boyutu çok büyük. Kodun büyüklüğü arttıkça ağaç yapısını gezmek daha fazla zaman alıyor.

Şuan için kısıtlı sayıda kusur için kontrol yapılıyor. Bunların sayısı arttırılabilir.

## 8. KAYNAKLAR

- [1] ANTLR, <http://www.antlr.org/>
- [2] Terence Parr, *The Definitive ANTLR 4 Reference*, The Pragmatic Programmers, 2013.
- [3] Code smell, [https://en.wikipedia.org/wiki/Code\\_smell](https://en.wikipedia.org/wiki/Code_smell)
- [4] Static program analysis, [https://en.wikipedia.org/wiki/Static\\_program\\_analysis](https://en.wikipedia.org/wiki/Static_program_analysis)
- [5] OCLint, <http://oclint.org/>
- [6] Cppcheck, <https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/>
- [7] Shyam R. Chidamber, Chris F. Kemerer, “A Metrics Suite for Object Oriented Design”, IEEE Transactions on Software Engineering, vol. 20, No. 6, June 1994, pp.476-493.
- [8] A. Kalay, “Statik Kod Analizinin Yazılım Geliştirme Sürecindeki Yeri ve Yazılım Kalitesine Etkisi”, 4. ULUSAL YAZILIM MÜHENDİSLİĞİ SEMPOZYUMU, UYMS, 2009, pp.211-218.
- [9] M. Rizwan Jameel Quresh, Waseem A. Qureshi, “Evaluation of the Design Metric to Reduce the Number of Defects in Software Development”, Saudi Arabia.
- [10] Cohesion metrics, <http://www.aivosto.com/project/help/pm-oo-cohesion.html>
- [11] NDepend Code Metrics Definitions, <http://www.ndepend.com/docs/code-metrics#LCOM>
- [12] Chidamber & Kemerer object-oriented metrics suite, <http://www.aivosto.com/project/help/pm-oo-ck.html>
- [13] Feza Buzluca, “BLG 625 - Yazılım Tasarımı Kalitesi Ders Notları”, 2014, <http://ninova.itu.edu.tr/tr/dersler/fen-bilimleri-enstitusu/2716/blg-625/ekkaynaklar?g403200>
- [14] Refactoring, <https://sourcemaking.com/refactoring>
- [15] FindBugs Bug Description, <http://findbugs.sourceforge.net/bugDescriptions.html>
- [16] Code Smell, <http://c2.com/cgi/wiki?CodeSmell>
- [17] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
- [18] Software metric, [https://en.wikipedia.org/wiki/Software\\_metric](https://en.wikipedia.org/wiki/Software_metric)
- [19] List of tools for static code analysis, [https://en.wikipedia.org/wiki/List\\_of\\_tools\\_for\\_static\\_code\\_analysis](https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)
- [20] Clang, <http://clang.llvm.org/>
- [21] Standard C++, “Working Draft, Standard for Programming Language C++”, 2015, <https://isocpp.org/>

# EKLER

## EK 1. Kusurlu Kod Listesi [3], [5], [6], [14], [15], [16], [17]

1. İsimlendirme kurallarına uyulmaması. (bkz. Naming conventions, Style guidelines)
2. Yorum ve açıklama yazılmaması. (bkz. Comments)
3. Büyük sınıflar. Bir sınıf bir nesneyi betimlemeli, sadece bir sorumluluğu olmalıdır. Farklı işler yapmamalıdır. (bkz. Single responsibility principle)
4. Veri saklayan sınıflar. Herhangi bir işi yoktur. Çok fazla alan (fields) içerir. (bkz. Data class)
5. Bir sınıfın işiyle ilgili çok az şey yapması. (bkz. Lazy class)
6. Sınıf içerisinde belli durumlarda kullanılan alanlar. (bkz. Temporary field)
7. Alt sınıf taban sınıfın veri ve methodlarının hepsini kullanmıyor. (bkz. Refused bequest)
8. Bir sınıfta değişiklik yapıldığında, sınıfın ilgisiz yerlerinde de değişiklik yapılması. (bkz. Divergent change)
9. Bir sınıfta değişiklik yapıldığında, birçok farklı sınıfta da değişiklik yapılması. (bkz. Shotgun surgery)
10. Bir sınıfın başka bir sınıfın üyelerini aşırı kullanması. (bkz. Feature envy)
11. Bir sınıfın tüm işlerini başka sınıflara yaptırması. (bkz. Middle man)
12. Message chains

```
A a;  
a.b().c().d().e().f().g().h().i();
```

13. Inappropriate Intimacy.
14. Primitive Obsession.
15. Data Clumps.
16. Alternative Classes with Different Interfaces.
17. Parallel Inheritance Hierarchies.
18. Incomplete Library Class.
19. Speculative Generality.
20. Combinatorial Explosion.
21. Indecent Exposure.
22. Downcasting.
23. Cyclomatic complexity.
24. Üst sınıf, alt sınıfa bağlı.
25. Uzun fonksiyon. Her fonksiyon sadece tek bir iş yapmalıdır. Fonksiyonların kısa olması okuma ve anlaşılmayı kolaylaştırır.
26. Methodların fazla parametreye sahip olması.

**27.** Çok sayıda kod bloğunun iç içe yuvalanması.

```
if (true)
{
    if (true)
    {
        if (true)
        {
            if (true)
            {
                if (true)
                {
                    // ...
                }
            }
        }
    }
}
```

**28.** Hardcoded sayı ve kelimeler. (bkz. Magic numbers)

**29.** Kod tekrarı (bkz. Duplicated code)

**30.** Kullanılmayan değişkenler.

**31.** Kullanılmayan method parametreleri.

**32.** Kullanılmayan methodlar, sınıflar. (bkz. Dead code)

**33.** “goto” deyimini kullanmak.

```
void f()
{
    label:
        // do something

        goto label;
}
```

**34.** == (eşit operatörü) yerine = (atama operatörünü) kullanmak

```
if (i = 0)
{
    // do something
}
```

**35.** “switch” deyiminin “break” keywordünü içermemesi.

```
switch (i)
{
    case 1:
        break;
    case 2:
        // do something
    default:
        break;
}
```



**36. void pointer.**

```
int n;
float f;

void* ptr;
ptr = &n; // valid
ptr = &f; // valid
```

**37. Erişilemeyen kod.**

```
int f()
{
    // do something

    return 1;

    int i;
}
```

**38. Macro yerine inline function, enum, const kullanılmalı.****39. Boş kod blokları. (if, else, for, while, do/while, switch, try, catch, finally)**

```
if (condition)
{
}

if (condition)
{
    // do something
}
else
{
}

for (int i = 0; i < length; i++)
{
}
```

**40. Methodların içerisinde parametrelere tekrardan değer atamak.**

```
void f(int i)
{
    // do something
    i = 0;
}
```

**41. Yerel bir değişkenin referansını döndürmek.**

```
int& f()
{
    int i;
    return i;
}
```

**42. Yanlış “;” (noktalı virgül) kullanımı**

```
if (true);
    DoSomething();

while (true);
    DoSomething();
```

```

for (size_t i = 0; i < length; i++);
    DoSomething();

int i;;
DoSomething();;
;

```

#### 43. Referans veya işaretçilerin tip dönüşümleri.

```

class A
{
    /* ... */
};

class B
{
    /* ... */
};

A * a = new A;
B * b = reinterpret_cast<B*>(a);

```

```

class A
{
    /* ... */
};

class B : A
{
    /* ... */
};

A a;
B& b = dynamic_cast<B&>(a);

```

#### 44. Dinamik belleklerin serbest bırakılmaması, dosyaların kapatılmaması, vb.

```

int* i;
//do something
//delete i;

fstream file;
file.open("test.txt");
//do something
//file.close();

```

#### 45. Exceptions (istisnai durum) yutulmamalı. Sadece işlenebilecekse yakalanmalı.

#### 46. Spaghetti code.

#### 47. Satırların uzun olması. Ekrana sığmaması. Okumayı zorlaştırır.

#### 48. Değişkenler, fonksiyonlar, sınıflar anlamlı isimlere sahip olmalı.

#### 49. Sonsuz döngü.