SOFTWARE ENGINEERING

Week 10

Software Quality

Dr. A. Cüneyd TANTUĞ Dr. Tolga OVATMAN
Istanbul Technical University
Computer Engineering Department

Agenda

1. Software Quality and Quality Engineering
2. Object Oriented Metrics
3. Software Design Quality
   1. Design Principles
   2. Desing Anti-Patterns
   3. Defect Symptoms
4. Design Patterns

1. Software Quality and Quality Engineering
2. Object Oriented Metrics
3. Software Design Quality
   1. Design Principles
   2. Desing Anti-Patterns
   3. Defect Symptoms
4. Design Patterns

Software Quality and Quality Engineering

10.1

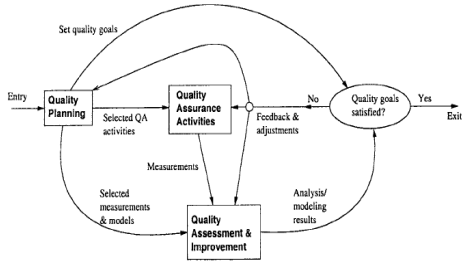
Software Quality

- Most software quality assessment approaches adopt two major criteria
  - Software should do the right thing
  - Software should do it right
- ISO standard categorizes software quality assessment crtieria
  - Functionality
  - Reliability
  - Usability
  - Effectiveness
  - Maintainability
  - Portability

Software Quality

- Kitchenham and Pfleeger proposes diffferent perspectives for software quality assessment
  - Transcendental view: Quality is hard to define or describe in abstract terms, but can be recognized if it is present. It is generally associated with some intangible properties that delight users.
  - User view: Quality is fitness for purpose or meeting user's needs.
  - Manufacturing view: Quality means conformance to process standards.
  - Product view: The focus is on inherent characteristics in the product itself in the hope that controlling these internal quality indicators will result in improved external product behavior (quality in use).
  - Value-based view: Quality is the customers' willingness to pay for a software

Quality Engineering



Quality Engineering

- Quality assurance and improvement practices
  - Measurement
  - Analysis and modeling
  - Feedback
  - Follow-up activities

1. Software Quality and Quality Engineering
2. Object Oriented Metrics
3. Software Design Quality
   1. Design Principles
   2. Desing Anti-Patterns
   3. Defect Symptoms
4. Design Patterns

Object Oriented Metrics

10.2

Measuring Software Quality

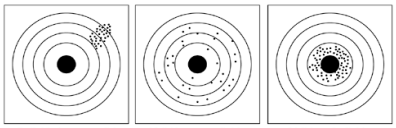
- Measurement is the basic principle of quality engineering. To reach conclusions on a scientific basis on software quality we need to have empirical data based on observations. The way to achieve this lies in measurement.
- By measurements we can set up a common platform on subjective and ambigious concepts.

"Design and code reviews are adequate for quality assurance"

- How do we understand if we've reached our target?

Measuring Software Quality

- Measurement quality is characterized by
  - Reliability
    - Smaller random effects
  - Geçerlilik
    - Larger and systematic effects



Reliable but not valid | Valid but not reliable | Valid *and* reliable

Object Oriented Metrics

- Lots of measurableswhich can be subject to object oriented metric derivation: class, methods, inheritance, ...
- Two major concepts in measurement:
  - Coupling
  - Cohesion
- For instance following questions are just a small sample on what can be inspected
  - Is it good for a class to have many methods?
  - Is it good for a class to have many variables?
  - Is it better for a class hierarchy to be wider on deeper in terms of inheritance?

Object Oriented Metrics

- Lorenz metric set is developed to adress such issues
  - Average method size (LOC) : For smalltalk <8, for C++ <24
  - Average number of methods per class: should be less than 20
  - Average number of cariables per class: should be less than 6
  - Class hierarchy depth: should be less than 6
  - Module to module interactions: should be less than 6
  - Class to class interactions: Should be considerably more than M2M
  - Comment lines per method: Should be larger than 1
  - Number of classes-methods deprecated: Should be monotonic

Object Oriented Metrics

- Chidamber and Kemerer metric set is the most widely accepted metric set:
  - WMC (Weighted Methods per Class) : Adjusted using cyclomatic complextiy
  - DIT (Depth of Inheritance Tree)
  - NOC (Number of Children of a Class)
  - CBO (Coupling Between Object Classes): Number of distinct methods/objects a class accesses
  - RFC (Response for a Class) : Average number of methods that execute due to an incoming message to an object
  - LCOM (Lack of Cohesion on Methods): Number of distinct methods in a class.

Object Oriented Metrics

- Basili et al. validated CK metric set on 180 Obj.O. classes developed by 8 different teams.
  - 6 metrics are uncorralated
  - Low DIT and NOC points to lack of inheritance usage
  - LCOM is not correlated to defective classes
  - DIT, RFC, NOC and CBO are highly correlated to defective classes
  - Better than older metric sets in detecting defective classes.

Object Oriented Metrics

- Managerial analysis show that higher CBO and LCOM values indicate a lower productivity, lower reuse and higher effort on design/coding:
  - Classes with higher CBO on average has 77 line/hour lower productivity
  - Classses with higher LCOM on average has 34 line/hour lower productivity
- Rosenberg, Stapko and Gallo(1999) showed that classes having more than one of the following indication have a higher defect rate:
  - RFC > 100
  - RFC > 5* number of methods
  - CBO > 5
  - WMC > 100
  - Number of methods > 40
- QMOOD/2 is one of the most widely used quality models based on OO metrics.

1. Software Quality and Quality Engineering
2. Object Oriented Metrics
3. Software Design Quality
   1. Design Principles
   2. Desing Anti-Patterns
   3. Defect Symptoms
4. Design Patterns

Software Design Quality

10.3

Design Principles

- To produce quality software it is good to consider some principles originated from past experience.
  - Separation of Concerns
  - Open-close principle
  - KISS – Good enough – YAGNI
  - Liskov's substitution principle
  - Design by Contract
  - Demeter rules
  - DRY – Don't repeat yourself
  - SRP – Single responsibility principle

Design Principles

- Liskov's substitution principle
  - Proposed by Barbara Liskov:
    - "Functions that use a pointer to a base class object should be directly able to access derived classes' object"
  - Violating this principle also in contrast to open-close principle.

Design Principles

- Let's consider the following class

```
class Rectangle
{
  public:
    void   SetWidth(double w)  {itsWidth=w;}
    void   SetHeight(double h) {itsHeight=w;}
    double GetHeight() const   {return itsHeight;}
    double GetWidth() const    {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};
```

- If we were to add a Square class, does it make sense to derive it from Rectangle?
- setWidth() and setHeight()?

```
void Square::SetWidth(double w)
{
  Rectangle::SetWidth(w);
  Rectangle::SetHeight(w);
}

void Square::SetHeight(double h)
{
  Rectangle::SetHeight(h);
  Rectangle::SetWidth(h);
}
```

Design Principles

- What about sending a square to a function that use a rectangle pointer?

```
void f(Rectangle& r)
{
  r.SetWidth(32); // calls Rectangle::SetWidth
}
```

- We can declare setWidth() and setHeight() as virtual to overcome this problem.
- Real problem is: 'Does Square really behave like a rectangle?'

```
void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

Design Principles

- Demeter principles
  - Proposed in Northeastern university. A.k.a "Don't talk to strangers" principle.
  - O object's M method should only be able to access the following:
    - O object
    - M's parameters
    - Objects constructed in M
    - Objects in O
    - Global variables accessible by O

Design Principles

- Consider the following snippet from a payment system

```
payment = 2.00;  // "I want my two dollars!"
Wallet theWallet = myCustomer.getWallet();
if (theWallet.getTotalMoney() > payment) {
    theWallet.subtractMoney(payment);
} else {
    // come back later and get my money
}
```

- Should we get the customer's wallet?
- Customer should be a delegate for the Wallet.

Anti-patterns

- Anti-patterns are defective design/code examples that emerge unwillingly. Some examples:
  - Interface bloat
  - Magic buttons
  - Magic numbers
  - Race conditions
  - Super calls
  - God class
  - Spaghetti code
  - Shotgun surgery

Anti-patterns

- Magic button: Emerges when business logic is coded in button action events. Following problems can occur:
  - Code in event handlers can grow unexpectadly.
  - Gets harder and harder to modify GUI
  - TeHarder to test and debug
- Commonly seen in programmers experienced in component based programming like DELPHI

```
procedure TForm1.Button1Click(Sender:    TObject);
var
  reg: TRegistry;
begin
  reg := TRegistry.Create;
  try
    reg.RootKey := HKey_Current_User;
    if reg.OpenKey('\Software\MyCompany',    true) then
    begin
      reg.WriteString('Filename',    Edit1.Text);
    end;
  finally
    reg.Free;
  end;
end;
```

Anti-patterns

- Using business objects is the cure

```
type
  TPreferences = class
  private
    FFilename: String;
    procedure SetFilename(const Value: String);
  public
    property Filename: String read FFilename write SetFilename;
    procedure Load;
    procedure Save;
  end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  Preferences.Save;
end;

procedure TForm1.Edit1Change(Sender: TObject);
begin
  Preferences.Filename := Edit1.Text;
end;
```

Anti-patterns

- Magic numbers: Emerges when numbers are used without indicating their meaning directly at coding time.

```
for i from 1 to 52
   j := i + randomInt(53 - i) - 1
   a.swapEntries(i, j)
```

- instead

```
constant int deckSize := 52
for i from 1 to deckSize
   j := i + randomInt(deckSize + 1 - i) - 1
   a.swapEntries(i, j)
```

- Hence:
  - Code becomes easier to understand
  - Value of constant can be modified at a single point
  - More parametric code
  - Typos can be detected by compiler

Defect Symptoms

- Sometimes named code smells or bad smells; defect symptoms are weird coding practices that continously emerges when a deeper defect is present in the system.
- Some defect symptoms
  - Code repetition
  - Feature envy
  - Weird switches
  - Parallel inheritance
  - Empty catches
  - Using instanceof in decisions

Defect Symptoms

- Code repetition: Slightly different code being used repetitively in many different places.
  - Repetition of similar methods: Polymorphism is the remedy
  - Repetition of code blocks: Lambda functions are remedy
  - Repetition of code snippets: Aspects are remedy

Defect Symptoms

- Feature Envy: A class is heavily using another class' methods. Might not always be a bad situation when planned. Otherwise be aware

```
private Address currentAddress = null;
public string MailingAddress()
{
   StringBuilder sb = new StringBuilder ();
   sb.Append(currentAddress.AddressLine1);
   sb.Append("\n");
   sb.Append(currentAddress.City + "," + currentAddress.State);
   sb.Append("\n");
   sb.Append(currentAddress.PostalCode);
   return sb.ToString();
}
```

- This situation is not very maintainable and can be subject to unecessary code repetition.

```
private Address currentAddress = null;
    public string MailingAddress()
    {
      return currentAddress.ToString() ;
    }
```

Defect Symptoms

- Empty catches: Self explanatory. Exceptions should be rethrown at minimum.
- Using instanceof in decisions: It is meaningful to use instanceof operator only in equals method. Otherwise it is very unlikely that you may need it in a non-defective code.

```
public static void doSomething(Animal aAnimal){
  if (aAnimal instanceof Fish){
    Fish fish = (Fish)aAnimal;
    fish.swim();
  }
  else if (aAnimal instanceof Spider){
    Spider spider = (Spider)aAnimal;
    spider.crawl();
  }
}
```

```
public static void main(String... aArgs){
  Animal animal = new Animal();
  doSomething(animal);

  animal = new Fish();
  doSomething(animal);

  animal = new Spider();
  doSomething(animal);

  log("Done.");
}

public static void doSomething(Animal aAnimal){
  aAnimal.move();
}
```

Design Patterns

10.4

Design Patterns

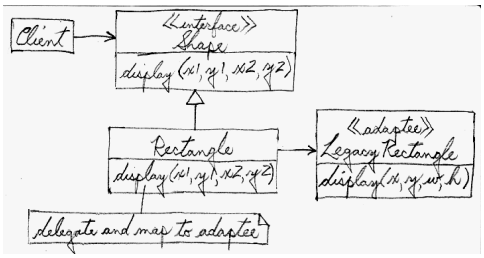
- GoF (Gang of Four) patterns was proposed in a book written in 1994:
- Gamma E., Helm R., Johnson R., Vlissides J.,
- Design Patterns : Elements of Reusable
- Object-Oriented Software , Reading MA,
- Addison-Wesley.
- 23 patterns are present in the book. GoF patterns are categorized in 3 groups:
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

Structural Patterns

Adapter

- **Problem**
  - An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.
- **Intent**
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Wrap an existing class with a new interface.
  - Impedance match an old component to a new system

Structural Patterns - Adapter



Before Adapter

```
class  LegacyLine   {
   public  void  draw(  int  x1,  int y1,  int  x2,  int y2  ) {
       System.out.pr in tln (    "line   from  (" + x1  + ',' + y1 + ") to
                                                 (" + x2  + ',' + y2 + ')'  );
}  }
class  LegacyRectang le    {
   public  void  draw(  int  x,  int  y,  int  w,  int  h  ) {
       System.out.pr in tln (    "rectangle   at  ("  + x + ',' + y
            + ") with  width  " + w + " and  height  " + h  );
}  }
public  class  AdapterDemo   {
   public  static  void  main(  String[]   args  ) {
       Object[]   shapes  = { new  LegacyLine(),
                              new  LegacyRectan gl e()    };
       // A begin  and  end  point  from  a  graphical   editor
       int x1  = 10, y1  = 20;
       int x2  = 30, y2  = 60;
       for  (int   i=0;  i  < shapes.leng th ;    ++i)
          if  (shapes[i].g et Cl as s() .g et Nam e( ).equ al s( "Lega cy Li ne" ))
              ((LegacyLine )s ha pe s[i ] ) .d raw (    x1,  y1,  x2,  y2  );
          else  if  (shapes[i]. ge tCl as s( ).g et Na me( ).equ al s( "Lega cy Re ct an gle " ) )
              ((LegacyRect an gl e) sha pe s[ i]) .d ra w(
                      Math.min(x1, x2) ,    Math.min(y1, y2 ),
                      Math.abs(x2- x1) ,    Math.abs(y2- y1 )    );
}  }
// line  from  (10,20)  to  (30,60)
// rectangle   at  (10,20)  with  width  20  and  height  40
```

After Adapter

```
class LegacyLine {
   public void draw( int x1, int y1, int x2, int y2 ) {
      System.out.println(  "line from (" + x1 + ',' + y1 + ") to
                                       (" + x2 + ',' + y2 + ')' );
}  }
class LegacyRectangle  {
   public void draw( int x, int y, int w, int h  ) {
      System.out.println(  "rectangle at (" + x + ',' + y
           + ") with width  " + w + " and height  " + h );
}  }
interface Shape {
   void draw(  int x1,  int y1, int  x2,  int  y2 );
}
class Line implements  Shape {
   private LegacyLine  adaptee = new LegacyLine();
   public void draw( int x1, int y1, int x2, int y2 ) {
      adaptee.draw(  x1, y1, x2, y2 );
}  }
```

After Adapter

```
class Rectangle implements Shape {
   private LegacyRectangle adaptee = new LegacyRectangle();
   public void draw( int x1, int y1, int x2, int y2 ) {
      adaptee.draw( Math.min(x1,x2), Math.min(y1,y2),
                    Math.abs(x2-x1), Math.abs(y2-y1) );
}  }
public class AdapterDemo {
   public static void main( String[] args ) {
      Shape[] shapes = { new Line(), new Rectangle() };
      // A begin and end point from a graphical editor
      int x1 = 10, y1 = 20;
      int x2 = 30, y2 = 60;
      for (int i=0; i < shapes.length; ++i)
         shapes[i].draw( x1, y1, x2, y2 );
}  }
// line from (10,20) to (30,60)
// rectangle at (10,20) with width 20 and height 40
```

Creational Patterns

Singleton

- Problem
  - Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.
- Intent
  - Ensure a class has only one instance, and provide a global point of access to it.
  - Encapsulated "just-in-time initialization" or "initialization on first use".

Creational Patterns - Singleton

Before Singleton

```
class GlobalClass {
   int m_value;
public:
   GlobalClass( int v=0 ) { m_value = v; }
   int get_value()        { return m_value; }
   void set_value( int v ) { m_value = v; }
};

// Default initialization
GlobalClass*  global_ptr = 0;

void foo( void ) {
   // Initialization on first use
   if ( ! global_ptr )
      global_ptr = new GlobalClass;
   global_ptr->set_value( 1 );
   cout << "foo: global_ptr is "
        << global_ptr->get_value() << '\n';
}

void bar( void ) {
   if ( ! global_ptr )
      global_ptr = new GlobalClass;
   global_ptr->set_value( 2 );
   cout << "bar: global_ptr is "
        << global_ptr->get_value() << '\n';
}
```

```
int main( void ) {
  if( ! global_ptr )
    global_ptr = new GlobalClass;
  cout << "main: global_ptr is "
       << global_ptr>get_value() << "\n";
  foo();
  bar();
}
// main: global_ptr is 0
// foo: global_ptr is 1
// bar: global_ptr is 2
```

After Singleton

```
class GlobalClass {
   int    m_value;
   static GlobalClass* s_instance;
   GlobalClass( int v=0 ) { m_value = v; }
public:
   int get_value()        { return m_value; }
   void set_value( int v ) { m_value = v; }
   static GlobalClass* instance() {
      if ( ! s_instance )
         s_instance = new GlobalClass;
      return s_instance;
   }
};

void foo( void ) {
   GlobalClass::instance()->set_value( 1 );
   cout << "foo: global_ptr is "
        << GlobalClass::instance()->get_value()
        << '\n';
}
```

After Singleton

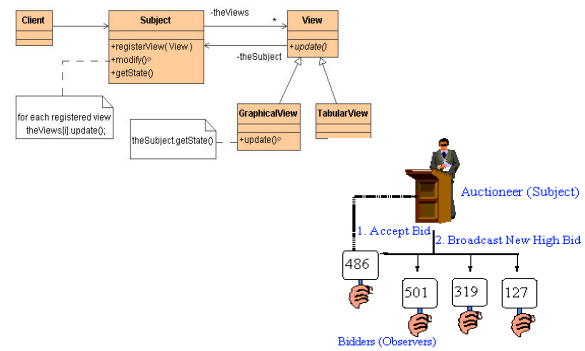
```
void bar( void ) {
   GlobalClass::instance()->set_value( 2 );
   cout << "bar: global_ptr is "
        << GlobalClass::instance()->get_value()
        << '\n';
}

int main( void ) {
   cout << "main: global_ptr is "
        << GlobalClass::instance()->get_value()
        << '\n';
   foo();
   bar();
}

// main: global_ptr is 0
// foo: global_ptr is 1
// bar: global_ptr is 2
```

Behavioral Patterns

Observer

- Problem
  - A large monolithic design does not scale well as new graphing or monitoring requirements are levied.
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
  - Encapsulate the core (or common or engine) components in a Subject abstraction, and the variable (or optional or user interface) components in an Observer hierarchy.
  - The "View" part of Model-View-Controller.

Behavioral Patterns - Observer



Before Observer

```
class DivObserver {
   int  m_div;
public:
   DivObserver( int div ) { m_div = div; }
   void update( int val ) {
      cout << val << " div " << m_div << " is "
           << val / m_div << '\n';
}  };

class ModObserver {
   int  m_mod;
public:
   ModObserver( int mod ) { m_mod = mod; }
   void update( int val ) {
      cout << val << " mod " << m_mod << " is "
           << val % m_mod << '\n';
}  };
```

Before Observer

```
class Subject {
   int  m_value;
   DivObserver  m_div_obj;
   ModObserver  m_mod_obj;
public:
   Subject() : m_div_obj(4), m_mod_obj(3) { }
   void set_value( int value ) {
      m_value = value;
      notify();
   }
   void notify() {
      m_div_obj.update( m_value );
      m_mod_obj.update( m_value );
}  };

int main( void ) {
   Subject  subj;
   subj.set_value( 14 );
}
```

After Observer

```
class Observer {
public:
   virtual void update( int value ) = 0;
};

class Subject {
   int  m_value;
   vector  m_views;
public:
   void attach( Observer* obs ) {
      m_views.push_back( obs );
   }
   void set_val( int value ) {
      m_value = value;  notify();
   }
   void notify() {
      for (int i=0; i < m_views.size(); ++i)
         m_views[i]->update( m_value );
}  };
```

After Observer

```
class DivObserver : public Observer {
   int  m_div;
public:
   DivObserver( Subject* model, int div ) {
      model->attach( this );
      m_div = div;
   }
   /* virtual */ void update( int v ) {
      cout << v << " div " << m_div << " is "
           << v / m_div << '\n';
}  };

class ModObserver : public Observer {
   int  m_mod;
public:
   ModObserver( Subject* model, int mod ) {
      model->attach( this );
      m_mod = mod;
   }
   /* virtual */ void update( int v ) {
      cout << v << " mod " << m_mod << " is "
           << v % m_mod << '\n';
}  };
```

After Observer

```
int main( void ) {
   Subject       subj;
   DivObserver   divObs1( &subj, 4 );
   DivObserver   divObs2( &subj, 3 );
   ModObserver   modObs3( &subj, 3 );
   subj.set_val( 14 );
}
```

Wrap-up

This week we present

- Software Quality Engineering
  - The process of assuring quality software
- Object Oriented Software Metrics
  - How do we measure quality of a software
- Design Principles and Design Patterns
  - Basic Body of Knowledge on producing Quality Software

Next Week

- You will be presenting your projects!!!