

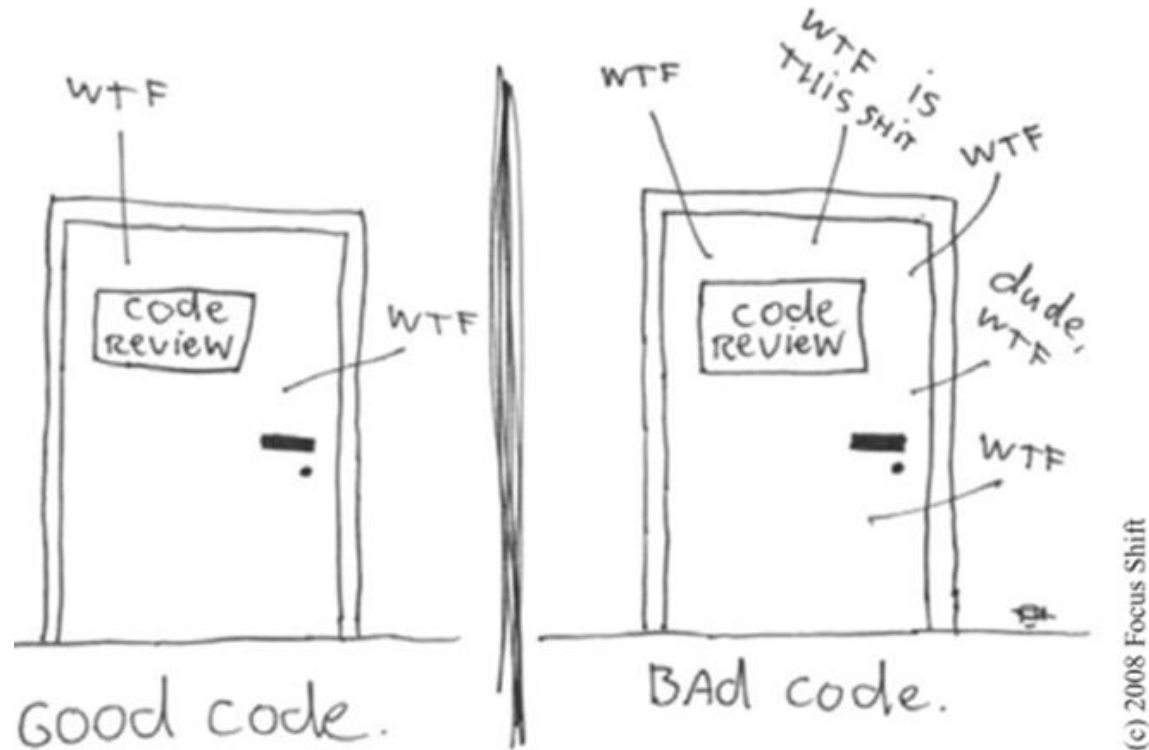


# Clean Code



# Übersicht

- Clean Code
  - Was ist Clean Code?
  - Praktiken (speziell: Benennungsregeln)
  - Prinzipien
- Metriken
- Code Review



Reproduced with the kind permission of Thom Holwerda.  
[http://www.osnews.com/story/19266/WTFs\\_m](http://www.osnews.com/story/19266/WTFs_m)

## Was ist Clean Code?

„Clean“ ist in erster Linie Quellcode aber auch Dokumente, Konzepte, Regeln und Verfahren, die intuitiv verständlich sind. Intuitiv verständlich ist alles, was mit wenig Aufwand in kurzer Zeit richtig verstanden werden kann.



# Was ist Clean Code?

- **Bjarne Stroustrup:**  
I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.
- **Dave Thomas:**  
Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone.



# Was ist Clean Code?

Clean Code: A Handbook of Agile Software Craftsmanship  
(Robert C. Martin)

## Prinzipien

- Don't repeat yourself (DRY)
- Keep it simple, stupid (KISS)
- Beware of Optimizations
- Favour Composition over Inheritance (FCoI)
- Single Responsibility Principle (SRP)
- Separation of Concerns (SoC)
- Coding Conventions
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Liskov Substitution Principle (LSP)
- Information Hiding Principle
- Principle of Least Astonishment
- Tell, don't ask
- Law of Demeter
- You ain't gonna need it (YAGNI)
- ...

## Praktiken

- Lesen
- Versionierung
- Scout Rule
- Root Cause Analysis
- Coding Conventions
- Documentation
- ...



# PRAKTIKEN



# Kleinigkeiten

- Lesen
  - Code lesen
  - Im Internet lesen (Artikel, How-Tos, Blogs, Foren, ...)
  - Bücher lesen
- Versionierung
  - SVN, Git, Mercurial, TFS, ...
  - Keine Angst vor Änderungen
  - Gemeinsam arbeiten



# Kleinigkeiten

- **Scout Rule**
  - Verlasse Code immer sauberer als du ihn betrittst.
  - Verfall beginnt mit Kleinigkeiten, die lang genug unbeachtet liegen bleiben.
- **Root Cause Analysis**
  - Nicht die Symptome sondern die Ursache bekämpfen.





# Coding Conventions

- Coding Conventions sind **Regeln** und **Richtlinien** für eine **bestimmte Programmiersprache**, die jeden Aspekt der Arbeit mit der Sprache behandeln.
- Coding Conventions behandeln z.B. ...
  - **Benennung** von Klassen, Methoden, Feldern, etc.
  - Datei- bzw. Projektstruktur
  - Struktur einzelner Dateien (Deklarationen, Funktionen, etc.)
  - Einrückungen, Klammerungen, White Spaces.
  - Art und Form von Kommentaren
  - Generische Sprach-Verwendung („Best practices“)



# Coding Conventions

- Zu **Beginn** des Projekts **festlegen**.
  - Hängen manchmal vom Kunden ab.
  - Bei uns: Fast ausschließlich **ReSharper-Regeln**
- **Vorteile:**
  - Einheitlicher Code
  - Erhöht Lesbarkeit
  - Weniger fehleranfällig
- **Nachteile:**
  - Manchmal ungewohnt
  - „religiöse“ Programmierer



# Benennungsregeln

- **Keine Abkürzungen**  
z.B. timeStamp statt tmp
- **Suchfreundliche Namen**  
z.B. days statt d
- **Keine humoristischen Namen**  
z.B. Deleteltems() statt DestroyEverything()
- **Ein Bezeichnung pro Konzept**  
z.B. Set[Name], Get[Name], Create[Name]...



# Benennungsregeln

- **Keine Artikel**  
z.B. BuddyList statt theBuddyList
- **Klassennamen:** Substantiv im Singular  
z.B. Customer
- **Methodennamen:** Verb (+ Substantiv)  
z.B. CreateCustomer, Create
- **Listen, Array, Dictionaries:** Plural  
z.B. List<Customer> customers



# Benennungsregeln

- **Kein Implementierungsmerkmale im Namen**  
z.B. nicht: CustomerSingleton
- **Keine technische Bestandteile**  
z.B. nicht: ClassNameImpl, ClassNameDelegate
- **Keine Weasel Words verwenden**
  - **Weasel Words** sind Wörter, die mehrere Bedeutungen haben bzw. deren Bedeutung stark vom Kontext abhängt.



# Weasel Words

AbstractDirectoryMonitor

ErrorCorrectingYakizakanaMigrator

SimpleGraphImporter

RunnableCommandQueue

ReadableStatementMediator

ExecutableNodeUtil

MultipleDeviceAdapter

MultipleMutexInitializer

StatefulConnectionInterpreter

AutomaticSocketBuffer

ThreadsafeRasterMonitor

AbstractKeystrokeMarshaller

ConfigurablePixelCollector

CryptographicThreadGenerator

ServiceManager

CustomerRepository



# PRINZIPIEN



# Don't repeat yourself (DRY)

- Dupliziere keinen Code.
- Duplizierter Code bedeutet:
  - Schlechtere Wartbarkeit
  - Änderungen in **jedem** Duplikat notwendig
  - Fehler werden meistens auch kopiert





# Don't repeat yourself (DRY)

Don't repeat yourself **verletzt**:

```
public interface IStringOutput {  
    public string StringOutput();  
}  
  
class ObjectOne : IStringOutput {  
    private string mString;  
  
    public string StringOutput() {  
        return mString;  
    }  
}
```

```
class ObjectTwo : IStringOutput {  
    private string mString;  
  
    public string StringOutput() {  
        return mString;  
    }  
}
```



# Don't repeat yourself (DRY)

Don't repeat yourself **eingehalten**:

```
public interface IStringOutput {  
    public string StringOutput();  
}
```

```
abstract class AbstractObject : IStringOutput {  
    private string mString;  
  
    public string StringOutput() {  
        return mString;  
    }  
}
```

```
class ObjectOne : AbstractObject {  
}
```

```
class ObjectTwo : AbstractObject {  
}
```



# Keep it simple, stupid (KISS)

- Löse Probleme so einfach wie möglich.
- Nicht die neuesten Sprachfeatures oder Technologien verwenden, nur weil man das mal ausprobieren will.
- Lieber **viele** sehr **einfache Methoden** als eine sehr komplizierte verwenden.

- **Nicht:**

```
k = i+++j;
```



# Beware of Optimizations

- Optimiere nicht.
- Für Experten: Optimiere noch nicht.
- Oft erreicht man mit Optimierung sein Ziel nicht.
- Optimierter Code ist ...
  - komplizierter.
  - anfälliger für Fehler.
  - schlechter lesbar.



# You ain't gonna need it (YAGNI)

- **Schreibe nur Code, der auch wirklich gebraucht wird.**
- Schreibe Code genau zu dem Zeitpunkt, an dem er gebraucht wird.
- Abgeleitet davon:
  - **Lösche** nicht verwendeten Code
  - Nicht auskommentieren, **löschen!**
- **Mehr Code** bedeutet **mehr Aufwand** bei Änderung der Architektur.
- Mehr Code wird nicht bezahlt.



# Favour Composition over Inheritance (FCoI)

- Leite nur ab, wenn du nicht komponieren kannst.
- Komposition ist **flexibler** als Ableitung.
- Komposition lässt sich **leichter ändern** und **testen**.
- Siehe **Vorlesung zur Architektur**.



# Tell, don't ask

- Sage einem Objekt, was es zu tun hat, anstatt den Zustand des Objekts abzufragen und dann zu entscheiden.
- Kleinere Schnittstellen
- Losere Kopplung



# Tell, don't ask

Tell, don't ask **verletzt**:

```
Hero hero = new Hero();  
Item item = new Item();  
  
int walletBalance = hero.Wallet.Balance;  
  
if (walletBalance >= item.Price) {  
    hero.GiveItem(item);  
    hero.Wallet.Balance = hero.Wallet.Balance - item.Price;  
}
```





# Tell, don't ask

Tell, don't ask **eingehalten**:

```
Item item = new Item();  
Hero hero = new Hero();  
int money = hero.BuyItem(item);  
  
if (money != item.Price) {  
    //...  
}
```



# Single Responsibility Principle (SRP)

- Klassen, Funktionen, Module usw. sollen nur für eine Aufgabe zuständig sein.
- Klare Verteilung der Aufgaben
- Kleine Klassen und Module



# Separation of Concerns (SoC)

- Teile das Programm so auf, dass einzelne Teile möglichst wenige Abhängigkeiten zur anderen Teilen haben.
- Auf **allen Ebenen** anwendbar (z.B. Projekt, Klassen, Methoden).
- **Projekt**
  - Mehrere Projekte mit eigenen Aufgaben.
- **Klassen**
  - Abhängigkeiten über **Interfaces** auflösen.
  - Nur eine Aufgabe pro Klasse.



# Separation of Concerns (SoC)





# Separation of Concerns (SoC)

- **Methodenebene**

z.B. `http://username:password@hostname/`

```
public string CreateAuthenticationUrl(string host) {  
    string username = GetUsername();  
    string password = GetPassword(username);  
  
    string authenticationUrl = CreateAuthenticationUrl(host, username, password);  
  
    return authenticationUrl;  
}
```



# Interface Segregation Principle (ISP)

- Schnittstellen sollen nur benötigte Funktionen veröffentlichen
- Möglichst kleine Interfaces
- So wenig Abhängigkeiten wie möglich



# Dependency Inversion Principle (DIP)

- Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen. Beide sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen



# Dependency Inversion Principle (DIP)

```
public class EncryptionService {  
    public void Encrypt(string sourceFileName,  
        string targetFileName) {  
        byte[] content;  
  
        using (var fs = new FileStream(sourceFileName,  
            FileMode.Open, FileAccess.Read)) {  
            content = new byte[fs.Length];  
            fs.Read(content, 0, content.Length);  
        }  
  
        byte[] encryptedContent = DoEncryption(content);  
  
        using (var fs = new FileStream(targetFileName,  
            FileMode.CreateNew, FileAccess.ReadWrite)) {  
            fs.Write(encryptedContent, 0,  
                encryptedContent.Length);  
        }  
    }  
}
```

```
private byte[] DoEncryption(byte[] content) {  
    byte[] encryptedContent = null;  
  
    // some encryption algorithm...  
    return encryptedContent;  
}
```





# Dependency Inversion Principle (DIP)

```
public interface IReader {  
    byte[] ReadAll();  
}  
  
public interface IWrite {  
    void Write(byte[] content);  
}  
  
public class EncryptionService {  
    public void Encrypt(IReader reader, IWriter writer) {  
        // Read content  
        byte[] content = reader.ReadAll();  
  
        // Encrypt  
        byte[] encryptedContent = DoEncryption(content);  
  
        // Write encrypted content  
        writer.Write(encryptedContent);  
    }  
}
```



# Dependency Inversion Principle (DIP)



# Liskov Substitution Principle (LSP)

- Ein Subtyp darf die Funktionalität eines Basistyps lediglich erweitern, aber nicht einschränken.
- Subtypen müssen sich also wie ihr Basistypen verhalten



# Liskov Substitution Principle (LSP)

Liskov Substitution Principle **verletzt**:

```
class Rectangle {  
    protected int mWidth;  
    protected int mHeight;  
  
    public virtual int Width {  
        get { return mWidth; }  
        set { mWidth = value; }  
    }  
  
    public virtual int Height {  
        get { return mHeight; }  
        set { mHeight = value; }  
    }  
  
    public int Area {  
        get { return mWidth * mHeight; }  
    }  
}
```

```
class Square : Rectangle {  
    public override int Width {  
        set {  
            mWidth = value;  
            mHeight = value;  
        }  
    }  
  
    public override int Height {  
        set {  
            mHeight = value;  
            mWidth = value;  
        }  
    }  
}
```



# Information Hiding Principle

- **Veröffentliche nur notwendige Methoden und Felder.**
- Verbergen erleichtert die Nutzung der Schnittstelle.
- Reduziert Abhängigkeiten.
- Erleichtert die Wartbarkeit der Klasse.



# Information Hiding Principle

## Information Hiding Principle

verletzt:

```
class EvenNumberContainer {  
    public int mSomeInternalValue;  
  
    public EvenNumberContainer() {  
        mSomeInternalValue = 1;  
    }  
  
    public void IncreaseNumber(){  
        mSomeInternalValue *= 2;  
    }  
}
```

## Information Hiding Principle

eingehalten:

```
class EvenNumberContainer {  
    private int mSomeInternalValue;  
  
    public EvenNumberContainer() {  
        mSomeInternalValue = 1;  
    }  
  
    public void IncreaseNumber() {  
        mSomeInternalValue *= 2;  
    }  
  
    public int Val {  
        get { return mSomeInternalValue; }  
    }  
}
```



# Principle of Least Astonishment

- Programmteile sollten sich verhalten wie erwartet.
- Keine **Nebenwirkungen**.
- Aus dem Namen einer Funktion sollte die Funktionalität klar erkennbar sein.

- **Nicht:**

```
class ElementList {  
    private List<Element> mElements;  
  
    public void Add(Element element) {  
        this.mElements.Remove(element);  
    }  
}
```



# Law of Demeter (LoD)

- Lange Abhängigkeitsketten zwischen den Klassen vermeiden.  
z.B.  $x = a.B().C().D();$
- Erschwert Wartung, Erweiterung und Tests
- Ok sind:
  - Methoden und Felder der **eigenen Klasse**
  - Methoden und Felder von **Parametern**
  - Methoden und Felder von **assoziierten Klassen**
  - Methoden und Felder von **lokal erzeugten Objekten**
  - Methoden und Felder **globaler Objekte**





# Law of Demeter (LoD)

Law of Demeter **verletzt**:

```
Hero hero = new Hero();  
Item item = new Item();  
  
int walletBalance = hero.Wallet.Balance;  
  
if (walletBalance >= item.Price) {  
    hero.GiveItem(item);  
    hero.Wallet.Balance = hero.Wallet.Balance - item.Price;  
}
```



# SOFTWAREMETRIKEN



# Softwaremetriken

- **Metriken** sind z.B.
  - cm, km, m<sup>2</sup>, kg, km/h, ...
- Eine **Softwaremetrik** ist eine Funktion, die eine **Eigenschaft einer Software** auf eine **Zahl** abbildet.
- Softwaremetriken werden benutzt zum
  - **Vergleichen**
  - **Bewerten**
- Verschiedene **Tools** berechnen Metriken
  - Sonar, NDepend, Visual Studio, ...



# Lines of Code

- Anzahl an Zeilen einer Klasse bzw. des ganzen Projektes.
- **Logisch**  
Nur der eigentliche Code wird gezählt.
- **Physikalisch**  
Alle Zeilen (inkl. leere Zeilen, Kommentare, etc.).



# Cyclomatic Complexity (CC)

- CC bestimmt, wie komplex eine Methode oder Klasse ist.
- **Methode**  
Anzahl der Ausführungspfade (`if`, `while`, `switch`, ...)
- **Klassen**  
Durchschnittliche Komplexität aller Methoden oder Summe der Komplexität aller Methoden (je nach Tool).
- **Schwellenwert**  
Methoden mit  $CC < 15$  sind kompliziert, bis  $CC < 30$  aber ok



# Lack of cohesion in methods (LCOM)

- Beschreibt die **Kohäsion** einer **Klasse**
- **Berechnung**

$$\text{LCOM3}(c) = \frac{m(c) - \left( \frac{\sum m_f(c)}{f(c)} \right)}{m(c) \cdot f(c)}$$

$m(c)$  := Anzahl an Methoden in  $c$

$f(c)$  := Anzahl an Feldern in  $c$

$m_f(c)$  := Anzahl an Methoden in  $c$  die auf das Feld  $f$  in  $c$  zugreifen

- **Schwellenwert**
  - LCOM3 = 0 bedeutet maximale Kohäsion
  - LCOM3 = 1 bedeutet keine Kohäsion
  - LCOM3 > 1 bedeutet Dead Code oder Variablen, die nur von außen benutzt werden.
  - LCOM3 < 1 ok

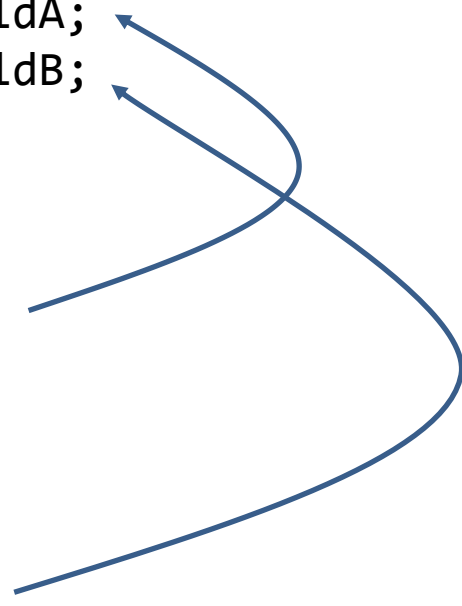


## Lack of cohesion in methods (LCOM)

```
class Class1
{
    private int mFieldA;
    private int mFieldB;

    void MethodA()
    {
        mFieldA = 1;
    }

    void MethodB()
    {
        mFieldB = 2;
    }
}
```



- $m(C1) = 1$
- $f(C1) = 1$
- $m_{mFA}(C1) = 1$
- $m_{mFB}(C1) = 1$
- $LCOM3(C1) = 1$



## Lack of cohesion in methods (LCOM)

```
class Class1
{
    private int mFieldA;

    void MethodA()
    {
        mFieldA = 1;
    }
}
```

```
class Class2
{
    private int mFieldB;

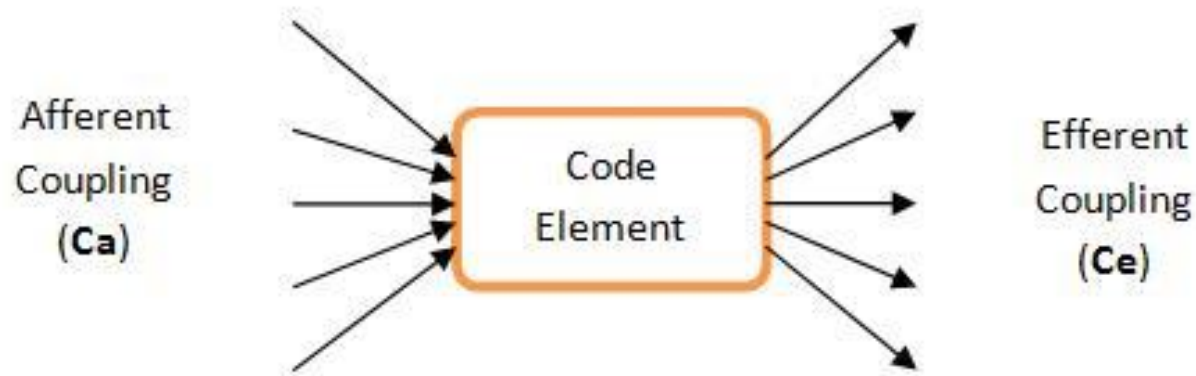
    void MethodB()
    {
        mFieldB = 2;
    }
}
```





# Coupling

- Beschreibt wie viele **Typen** von einer Klasse (oder Modul, Namespace, Assembly) **abhängen** oder umgekehrt.



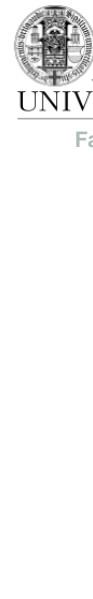


# CODE REVIEW

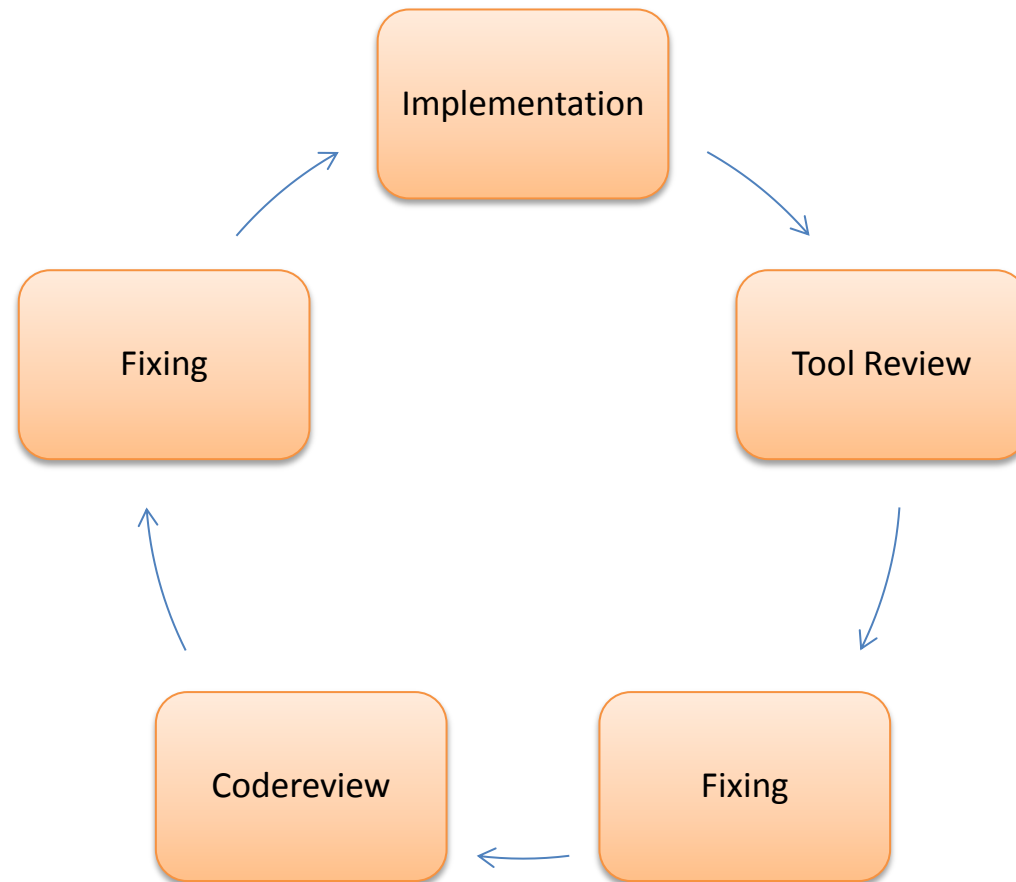


# Arten von Reviews

- „Manuelles“ Review
  - Pair Programming
  - Over-the-shoulder
  - Meetings
  - Mailinglisten, Foren, o.ä.
- Toolgestützte Reviews
  - Serverseitig: „Continuous Integration“ (Jenkins, Sonar... )
  - Clientseitig: z.B. statische Analyse (Ndepend, Resharper)



# Review Prozess





# Richtlinien für manuelle Reviews

- Optimale **Codelänge** etwa **300 Lines/Review**
- **Dauer**: max. 60 Minuten
- **Protokoll** der Probleme/Fehler
- **Keine Vorstellung durch Entwickler**
- Bei mehr als 2 Personen sollte eine Person die **Moderation** übernehmen



# Code Review

- **Vorteile**
  - Auffinden von Bugs
  - Verbesserung Codequalität
  - Einarbeiten von neuen Mitarbeiter
  - Vermittlung von Wissen
  - Menschlicher Nutzen
- **Nachteile**
  - Zeit & Kosten
  - Kann Teambotiviation und Zusammenhalt beschädigen ("Dein Code ist schlecht")
  - Kann Verbesserungen behindern (wenn immer der Chef reviewed)



**FRAGEN?**



# Quellen

- Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 Edition, 2008.
- <http://www.clean-code-developer.de/>
- <http://www.ndepend.com/Metrics.aspx>