

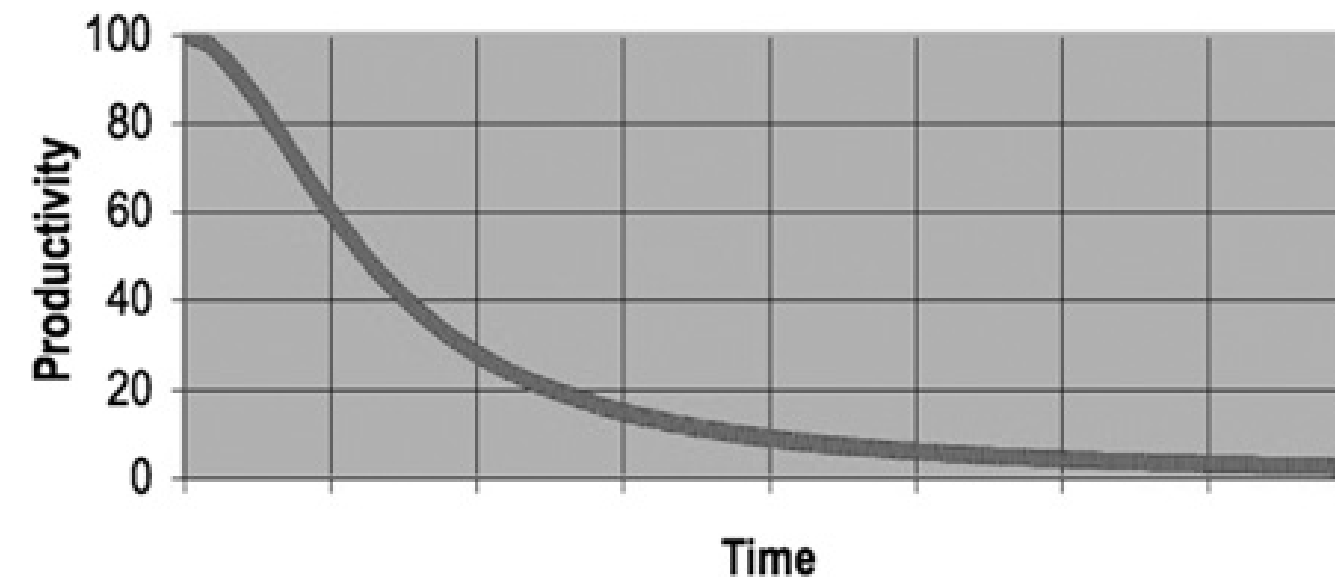
CLEAN CODE

WARUM IST CODE QUALITÄT WICHTIG?

- Programmieren ist hauptsächlich Implementierung von neuen Features
- Der Code kompiliert, macht was er soll, somit ist alles in Ordnung
- Fachbereiche bezahlen für Features, nicht für "schönen" Code

- Das Verhältnis von Code lesen zu schreiben ist 10:1 (!)
- Die meiste Zeit wird existierender Code gelesen und evtl. erweitert
  - Wartung ist die längste Phase des Produktlebenszyklus
  - Trifft auch zu, wenn ein komplett neues Modul geschrieben wird
- Du bist nicht allein!
  - Entwickler arbeiten in Teams

- Sinkende Produktivität über die Zeit bei unordentlichem Code



Quelle: Clean Code - A Handbook of Agile Software Craftsmanship (R.C. Martin)

# PROBLEME MIT CHAOTISCHEM CODE

- Schwer zu verstehen
  - und noch viel schwerer zu erweitern
- Neue Fehler schleichen sich bei Änderungen leichter ein
- Sinkende Produktivität
- Das Chaos wächst, wenn man nicht mit Umsicht handelt!

# BROKEN WINDOW THEORY


*A building with broken windows looks like nobody cares about it. So other people stop caring. They allow more windows to become broken. Eventually they actively break them.*

Dave Thomas and Andy Hunt

(Ursprünglich von den Sozialforscher James Q. Wilson and George L. Kelling, März 1982)


# DEFINITIONEN VON CLEAN CODE





*Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control.*

Grady Booch




*Clean code can be read, and  
enhanced by a developer other than  
its original author.*

Dave Thomas



*Clean code always looks like it was  
written by someone who cares.*

Michael Feathers



*Always code as if the guy who ends  
up maintaining your code will be a  
violent psychopath who knows  
where you live.*

Martin Golding

# WANN IST CODE CLEAN, SAUBER, AUFGERÄUMT?

- Einfach und direkt
- Aussagekräftige Namen
- Tut immer das, was man erwartet
- Einfach zu lesen und erweiterbar
- ...
- "Clean code does one thing well"

# DIE WERTE DES CLEAN CODE ENTWICKLERS

- Do Only What's Neccessary
- Isolate Aspects
- Minimize Dependencies
- Honor Pledges

# DO ONLY WHAT'S NECESSARY

- Vorsicht vor Optimierungen
- You Ain't Gonna Need It (**YAGNI**)
- Keep it simple, stupid (**KISS**)

# ISOLATE ASPECTS

- Don't Repeat Yourself (DRY)
- Separation of Concerns (SoC)
- Single Level of Abstraction (**SLA**)
- Single Responsibility Principle (**SRP**)
- Interface Segregation Principle (**ISP**)



# MINIMIZE DEPENDENCIES

- Information Hiding Principle
- **Law of Demeter**
- **Tell, don't ask**
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (**ISP**)
- Open Closed Principle (OCP)

# HONOR PLEDGES

- Überraschungen vermeiden
- **Principle of Least Astonishment**
- Implementation mirrors design
- Favour Composition over Inheritance (**FCoI**)
- Liskov Substitution Principle (LSP)

# THE BOY SCOUT RULE

Leave the campground cleaner than you found it!

# PRAKTIKEN DES CLEAN CODE ENTWICKLERS

- Embrace Uncertainty
- Focus
- Value Quality
- Get Things Done
- Stay Clean
- Keep Moving

# EMBRACE UNCERTAINTY

- Versionsverwaltung verwenden
- Automatisierte Unit- und Integrationstests
- Mit Mockups testen
- Continuous Integration einsetzen

# FOCUS

- Modular arbeiten
- Test first
- Limit WIP

# VALUE QUALITY

- Nur hohe Qualität akzeptieren
- Unittests automatisieren
- Code Reviews durchführen

# GET THINGS DONE

- Iterative Entwicklung
- Continuous Delivery
- Obergrenze für Work in Progress (WiP)



# STAY CLEAN

- Boy Scout Rule einhalten
- Regelmäßiges Refactoring
- Statische Code Analyse verwenden
- Code Coverage Analyse verwenden
- Coding Conventions einhalten

# KEEP MOVING

- Man lernt das ganze Leben
- Wissen verteilen
- Selbstreflexion
- Grundursachen bekämpfen, nicht die Symptome
- Ergebnisse messen
- Issue Tracking einsetzen
- Regelmäßige Retrospektiven im Team

# SOLID

- SRP: Single Responsibility
- OCP: Open/Close
- LSP: Liskov Substitution
- ISP: Interface Segregation
- DIP: Dependency Inversion


# SINGLE RESPONSIBILITY PRINCIPLE



*A class should have only one reason  
to change*

Robert C. Martin

# OPEN/CLOSE PRINCIPLE



*Software entities ... should be open for extension, but closed for modification.*


Bertran Meyer / Robert C. Martin

# LISKOV SUBSTITUTION PRINCIPLE

*Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*

Barbara Liskov

# INTERFACE SEGREGATION PRINCIPLE



*Many client-specific interfaces are better than one general-purpose interface.*

Robert C. Martin

# DEPENDENCY INVERSION PRINCIPLE



*Depend upon abstractions, not  
concretions.*

Robert C. Martin





# LITERATUR

- Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin
- Refactoring: Improving the Design of Existing Code. Martin Fowler und Kent Beck
- Working Effectively with Legacy Code. Micheal Feathers
- Clean Architecture: A Craftsman's Guide to Software Structure and Design. Robert C. Martin
- The Clean Coder: A Code of Conduct for Professional Programmers, Robert C. Martin
- The Pragmatic Programmer. From Journeyman to Master. Andrew Hunt und David Thomas