

# Parallele Code Smells: Eine Top 10 Liste

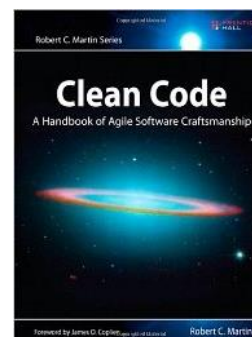
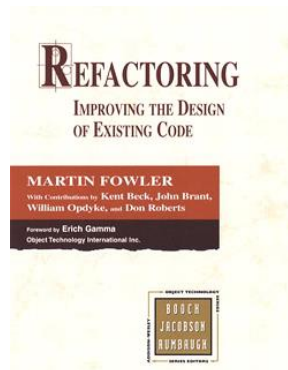
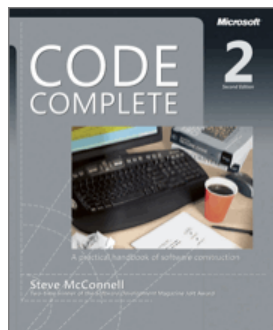
Luc Bläser

Hochschule für Technik Rapperswil

# Code Smells

---

- Symptome im Code
  - Hinweis für mögliche Designfehler
- Z.T. Kur durch Refactoring
  - Restrukturierung ohne Änderung des Verhaltens
- Bis anhin: Fokus auf sequentielles OO
  - Z.B. Riesen-Klasse, zu viele Parameter, Downcasts



# Parallele Code Smells

---

- Fokus auf Nebenläufigkeit und Parallelität
  - Am Beispiel für .NET und Java
  - Für andere Sprachen evtl. ebenso anwendbar
- Sammlung aus eigener Erfahrung
  - Aus Code Reviews in Industrie-Projekten
  - Zeitraum letzte 5 Jahre, nach Relevanz priorisiert

# Die Top 10 Liste

Frühere Präsentationen: Parallel 2016 Konferenz, Heise Developer Juli 2016  
(heute: erweiterte & überarbeitete Version)

# 1. Partly Synchronized Class

---

- Synchronisierte und nicht-synchronisierte aussen zugreifbare Member in derselben Klasse

```
class BankAccount {  
    private int balance;
```

```
    public int getBalance() { return balance; }
```

**unsynchronisiert**

```
    public synchronized void deposit(int amount) {  
        balance += amount;  
    }
```

**synchronisiert**

```
    public boolean withdraw(int amount) {  
        if (amount > balance) { return false; }  
        balance -= amount;  
        return true;  
    }
```

**unsynchronisiert**

```
}
```

Java

# Analog in .NET

---

```
class BankAccount {  
    private readonly object sync = new object();
```

```
    public int Balance { get; private set; } unsynchronisiert
```

```
    public void Deposit(int amount) {  
        lock (sync) {
```

**synchronisiert**

C#

```
        }  
    }
```

```
    public bool Withdraw(int amount) {  
        if (amount > Balance) return false;  
        Balance -= amount;  
        return true;  
    }
```

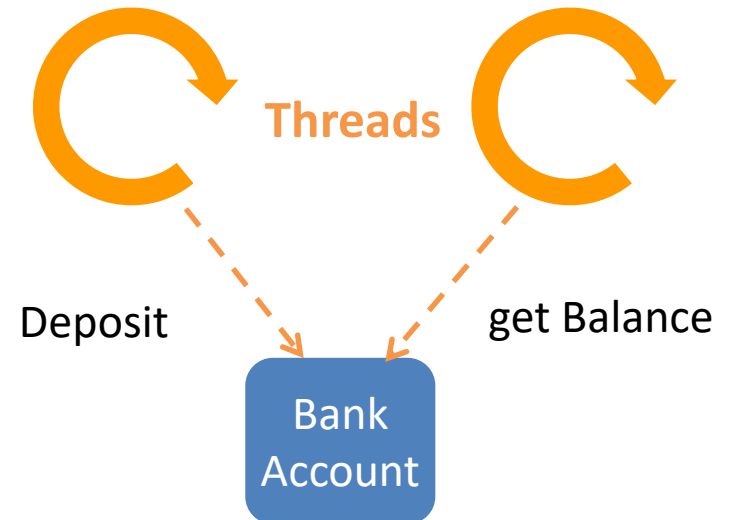
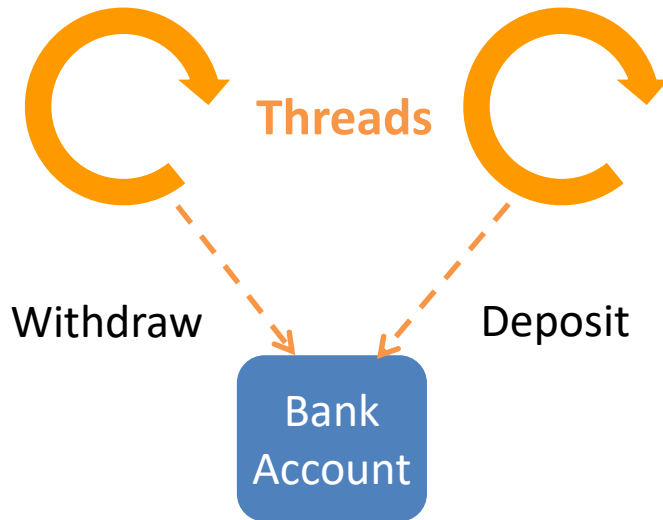
**unsynchronisiert**

```
}
```

# Problem: Halb Thread-Safe

---

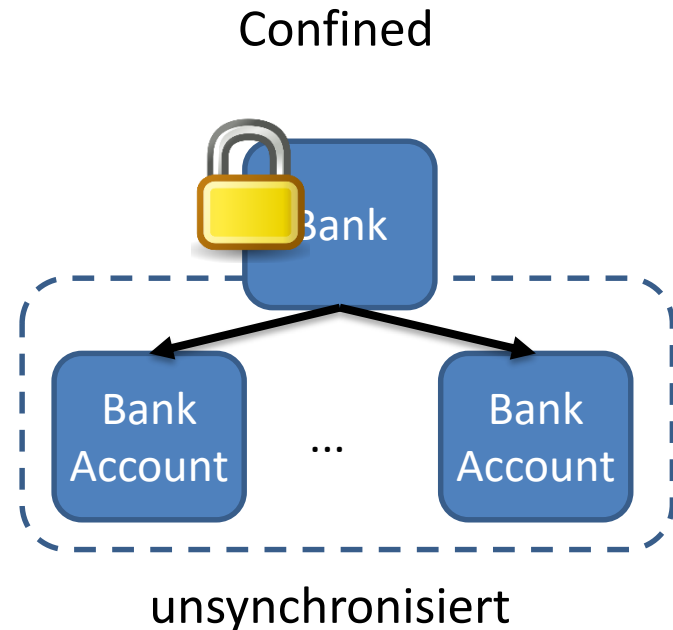
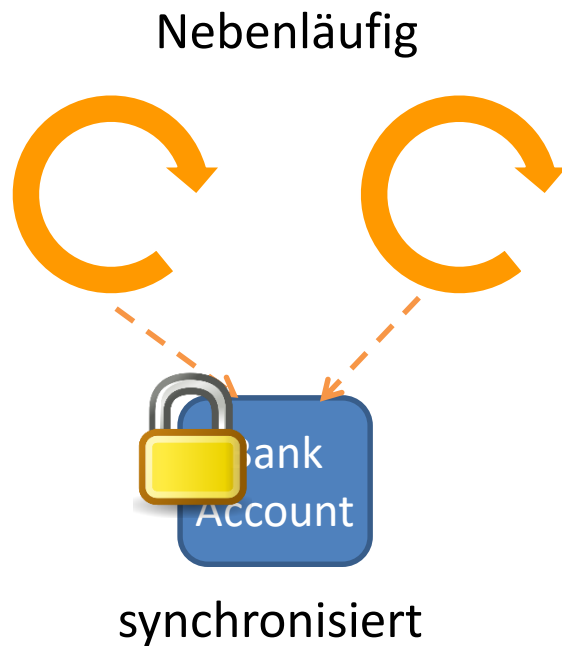
- Nur nebenläufige Deposit/Deposit sind sicher
- Andere Kombinationen nicht



**Data Races & Race Conditions**

# Kur: Klare Architektur

- Welche Threads verwenden welche Objekte?
- Definierte Verwendung pro Klasse/Objekt





## 2. Nested Locking Through Method Calls

---

- Synchronisierte Methode ruft direkt oder indirekt wiederum synchronisierte Methode auf

```
class BankAccount {  
    private int balance;  
  
    public synchronized void deposit(int amount) {  
        balance += amount;  
    }  
  
    public synchronized void transfer  
        (BankAccount target, int amount) {  
        balance -= amount;  
        target.deposit(amount);  
    }  
}
```

lock this  
lock target

Java

# Versteckte geschachtelte Locks

---

Thread 1

```
a.transfer(b, 10);
```

lock a  
lock b

Thread 2

```
b.transfer(a, 100);
```

lock b  
lock a

T1 sperrt a  
T2 sperrt b  
T1 will b  
T2 wil a



**Deadlock**

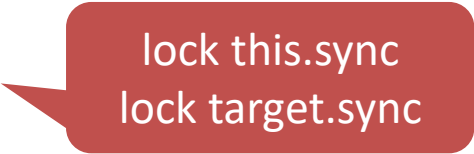
# Gleiches Problem in .NET

---

```
C# class BankAccount {
    private readonly object sync = new object();
    private int balance;

    public void Deposit(int amount) {
        lock (sync) { balance += amount; }
    }

    public void Transfer(BankAccount target, int amount) {
        lock (sync) {
            balance -= amount;
            target.Deposit(amount);
        }
    }
}
```



# Kur: Klare Architektur

---

- Wo werden Locks in welcher Reihenfolge bezogen?
- Geschachtelte Locks vermeiden
- Oder sonst lineare Sperrordnung

Lock [0] -----> Lock [2] --> Lock [3]



Konten nur nach  
aufsteigender  
Nummer sperren

# 3. Try-and-Fail Resource Acquisition

---

- Wiederholte Sperrversuche ohne Blockieren oder mit Timeout

```
a.acquire();  
while (!b.acquire(TIMEOUT)) {  
    a.release();  
    a.acquire();  
}
```



**Starvation**

Java

**Lösung: Blockierende Synchronisationsprimitiven vorziehen**

# 4. Use of Explicit Threads

---

- Starten von expliziten Threads

```
new Thread(() -> compute()).start();
```

Java



**Schlechte Skalierung:  
=> Zu viele Threads: Out of Memory**

# Kur: Tasks statt Threads

---

- Verwaltung über Thread Pool
  - Task = Potentiell parallel ausführbarer Arbeit
  - Begrenzte Anzahl Worker Threads
  - Skaliert leicht, rezykliert Threads

```
future = CompletableFuture.runAsync(() -> compute());
```

Java (Common Fork Join Pool)

```
task = Task.Run(Compute);
```

C# (.NET TPL)

# 5. Thread Pool Task Dependencies

---

- Tasks warten auf Bedingungen von anderen Tasks
  - Ausnahme: Warten auf Sub-Tasks ist okay

```
threadPool.submit(() -> {  
    condition.await();  
    ...  
});
```

Java

wartet auf

```
threadPool.submit(() -> {  
    ...  
    condition.signal();  
});
```

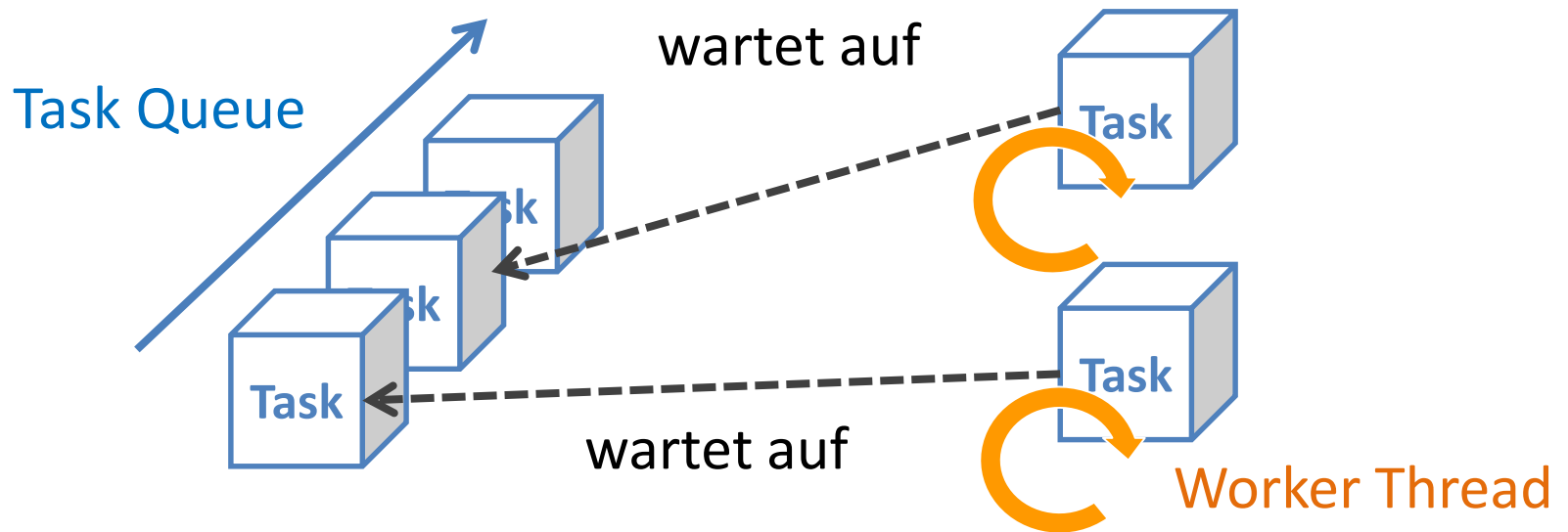


**Deadlock oder Skalierungsproblem**



# Task Warte-Abhängigkeiten

- Deadlock in Java: begrenzter Anzahl Worker Threads
- Ineffizient in .NET: TPL fügt langsam Threads hinzu



**Lösung: Task Continuations verwenden**

# 6. Fire and Forget

---

- Lancieren von Tasks, ohne später deren Ende oder Resultat abzuwarten

```
Java    CompletableFuture.runAsync(() -> {  
        ...  
    });
```

```
C#     Task.Run(() -> {  
        ...  
    });
```




**Diverse Probleme**

# Probleme bei Fire And Forget

---


- Exceptions im Task werden ignoriert

- In Java und .NET ab Version 4.5

```
CompletableFuture.runAsync(() -> {  
    ...  
    throw new RuntimeException();  ignoriert  
}
```

- Anwendung kann vor Task-Ende stoppen

- .NET TPL und Java ForkJoinPool benutzt Daemon Threads

```
CompletableFuture.runAsync(() => {  
    ...  
     plötzliches Ende  
    ...  
}
```

# 7. Uber-Asynchrony

---

- Grassierende Asynchronität bis in kleinste Methoden

```
async Task TranslateAsync() {  
    var input = await ReadAsync();  
    var output = await ProcessAsync(input);  
    await SaveAsync(output);  
}
```

C#

```
    async Task SaveAsync(Data data) {  
        foreach (var item in data) {  
            await InsertAsync(item);  
        }  
    }
```

```
        async Task InsertAsync(Item item) {  
            ...  
        }
```


# Unnötige Komplexität

---

- Unübersichtlich, viele Thread-Switches
- Grössere synchrone Logik, als Ganzes asynchron
  - Ausnahme, falls UI-Operationen in Methoden sind

`await Task.Run(Translate)`

```
void Translate() {  
    var input = Read();  
    var output = Process(input);  
    Save(output);  
}
```



```
void Save(Data data) {  
    foreach (var item in data) {  
        Insert(item);  
    }  
}
```

# 8. Monitor Single Wait / Single Signal

---

- Warten im Monitor ohne Schlaufe
- Einfaches Signal

```
synchronized(this) {  
    if (full) wait();  
    queue.add(x);  
    notify();  
}
```

Java

```
synchronized(this) {  
    if (empty) wait();  
    var x = queue.remove();  
    notify();  
}
```

# Typische Monitor-Fehler

---

- Wartebedingung immer wiederholt prüfen
  - `while (full) wait();`
  - Andere Threads können vor dem aufgeweckten Thread drankommen (Signal and Continue)
- Bei mehreren Wartebedingungen: Signal an alle
  - `notifyAll();`
  - Sonst wird evtl. nur ein Thread der falschen Bedingung geweckt (z.B. Wartender auf nicht-leer statt nicht-voll)
- Gleiches gilt in .NET!

# 9. Atomic, Volatile and Yield

---

- Atomare Instruktionen
- Volatile Variablen
- Thread Yield, Spin-Locks

```
var value = balance;  
if (value >= amount) {  
    Interlocked.Add(ref balance, -amount);  
}
```

C#



# Lock-freie Programmierung

---

- Komplex, fehleranfällig, oft ineffizient
  - Memory Model Expertise ist zwingend
- Unnötig in Applikationssoftware
  - Ausnahme: Low-level Algorithmen/Datenstrukturen

Lesen ohne  
Memory Barrier

```
var value = balance;  
if (value >= amount) {  
    Interlocked.Add(ref balance, -amount);  
}
```

if und Add sind  
nicht atomar



**Falsch**

# 10. Finalizers Accessing Shared State

---

- Finalizers mit Zugriff auf gemeinsame Ressourcen

```
C#  
public class Block {  
    public Block() {  
        Cache.NofBlocks++;  
    }  
  
    ~Block() {  
        Cache.NofBlocks--;  
    }  
}
```

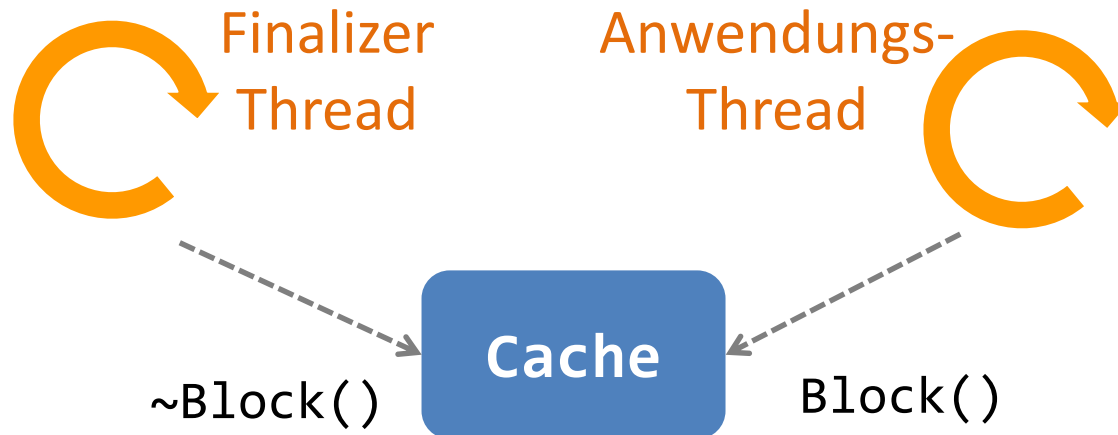


**Data Races & Race Conditions**

# Analyse: Finalizer

---

- Finalizer laufen nebenläufig zur Anwendung
- Saubere Synchronisation ist nötig



# Schlussfolgerungen

---

- Code Smells für parallele Aspekte
  - Sensibilisierung auf häufige Design-Fehler
- Beispiele anhand von Java und .NET
  - Allgemein gleiche Probleme in anderen Sprachen
- Es gibt weitere Code Smells
  - Jeder kann weitersammeln
- Kein Absolutismus
  - Nicht jeder Smell zeigt einen Fehler
- Idee: Code-Analyse Tool
  - Zur Erkennung solcher Smells

# Danke für Ihre Aufmerksamkeit

---

## ■ Kontakt

- **Prof. Dr. Luc Bläser**  
**HSR Hochschule für Technik Rapperswil**  
[lblaeser@hsr.ch](mailto:lblaeser@hsr.ch)
- **HSR Concurrency Lab**
  - <http://concurrency.ch>
- **Microsoft Innovation Center Rapperswil**
  - <http://msic.ch>