Parallele Code Smells: Eine Sammlung für .NET

Luc Bläser

Hochschule für Technik Rapperswil

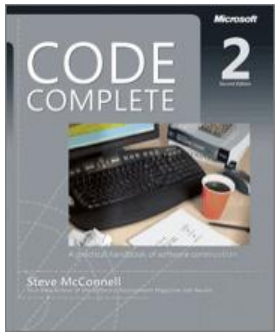
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

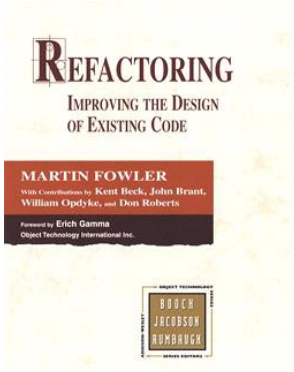
INSTITUT
FÜR
SOFTWARE

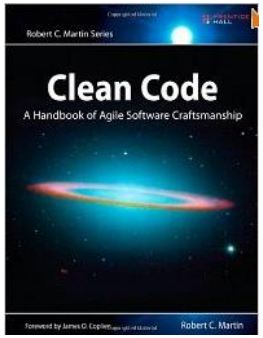
Parallel 2016 Konferenz
7. Apr. 2016, Heidelberg

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, ebenso für Java u.a. anwendbar
- Sammlung aus eigener Erfahrung
  - Aus Code Reviews in Industrie-Projekten
  - Zeitraum letzte 5 Jahre, nach Relevanz priorisiert

Top 10 Code Smells

1. Partly synchronized class
2. Nested locking through method calls
3. Try-and-fail resource acquisition
4. Use of explicit threads
5. Thread pool task dependencies
6. Fire-and-forget launches
7. Uber-asynchrony
8. Monitor single wait / single signal
9. Atomic, volatile and yield
10. Finalizers accessing shared state

1. Partly Synchronized Class

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

```
class BankAccount {
   public int Balance { get; private set; }

   public void Deposit(int amount) {
      lock (this) {
          Balance += amount;
      }
   }

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

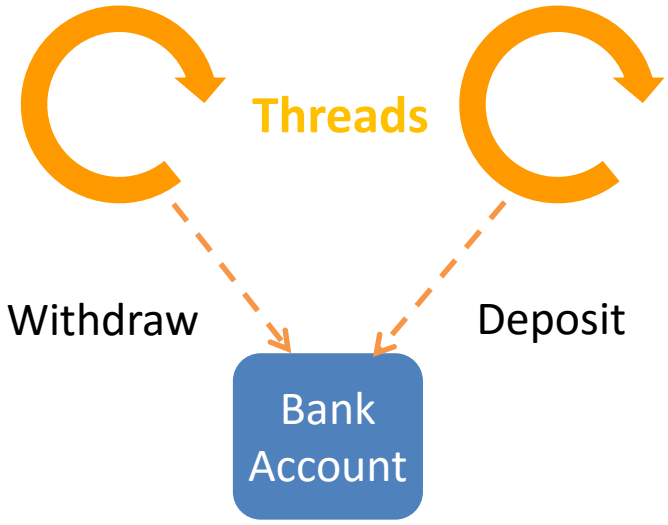
unsynchronisiert

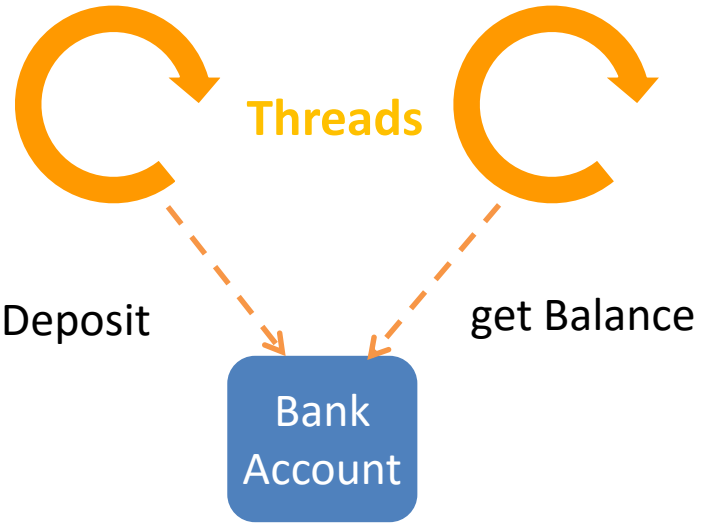
synchronisiert

unsynchronisiert

Problem: Halb Thread-Safe

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

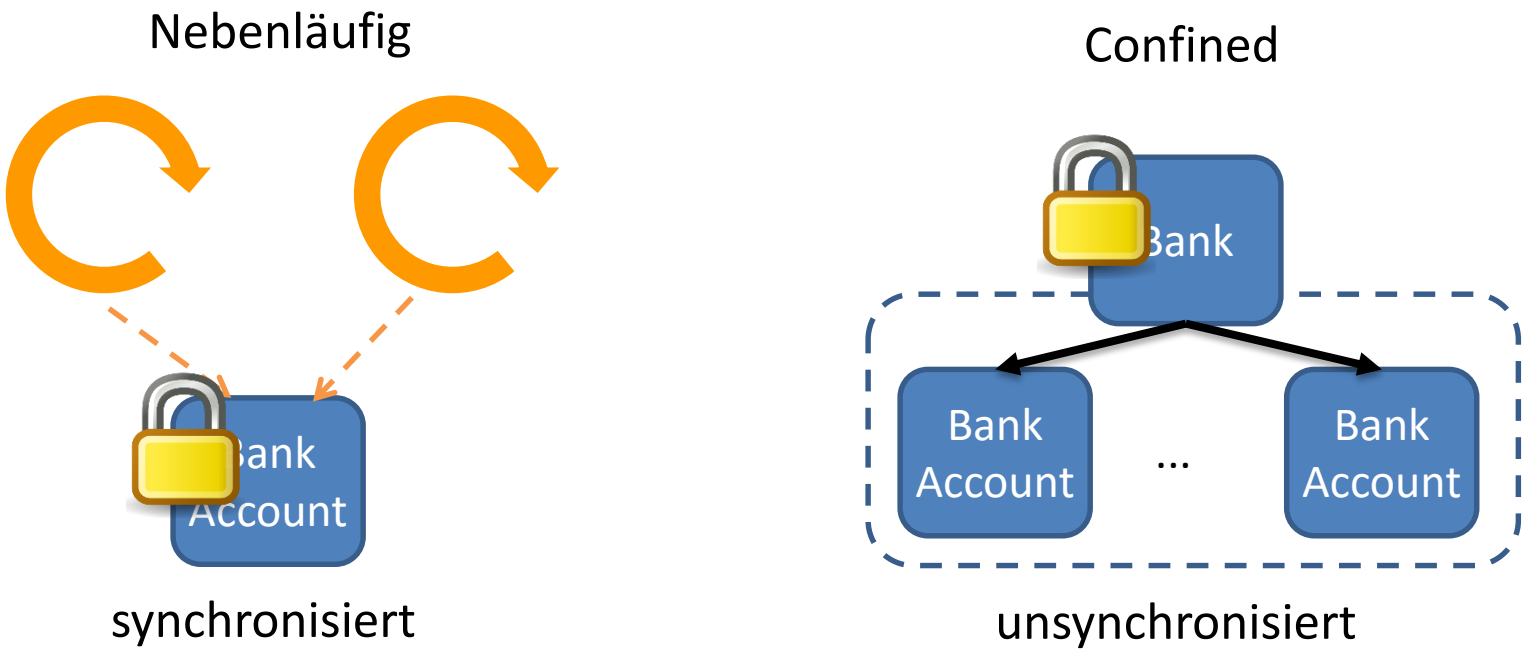




Race Conditions

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 void Deposit(int amount) {
      lock (this) { _balance += amount; }
   }

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

lock this
lock target

Versteckte geschachtelte Locks

lock a
lock b

Thread 1

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

lock b
lock a

Thread 2

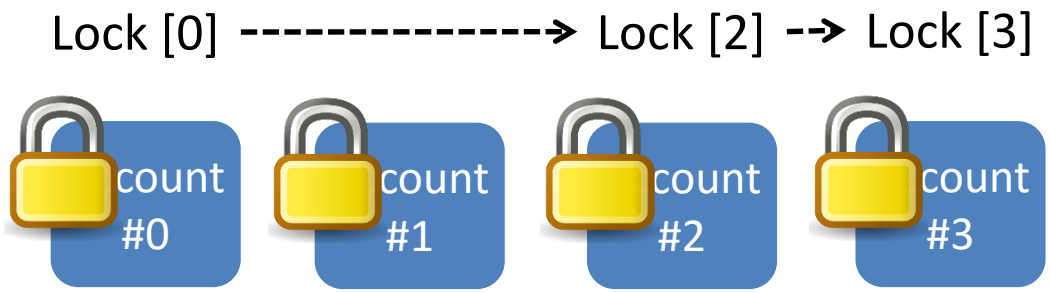
```
b.Transfer(a, 100);
```

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

Deadlock

Kur: Klare Architektur

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



Konten nur nach aufsteigender Nummer sperren

3. Try-and-Fail Resource Acquisition

- Wiederholte Sperrversuche ohne Blockieren oder mit Timeout

```
a.Wait();
while (!b.Wait(Timeout)) {
  a.Release();
  a.Wait();
}
```

Starvation

Lösung: Blockierende Synchronisationsprimitiven vorziehen

4. Use of Explicit Threads

- Starten von expliziten Threads

```
new Thread(Compute).Start();
```

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

Kur: Tasks statt Threads

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

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

5. Thread Pool Task Dependencies

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

```
Task.Run(() => {
  signal.Wait();
  ...
});
```

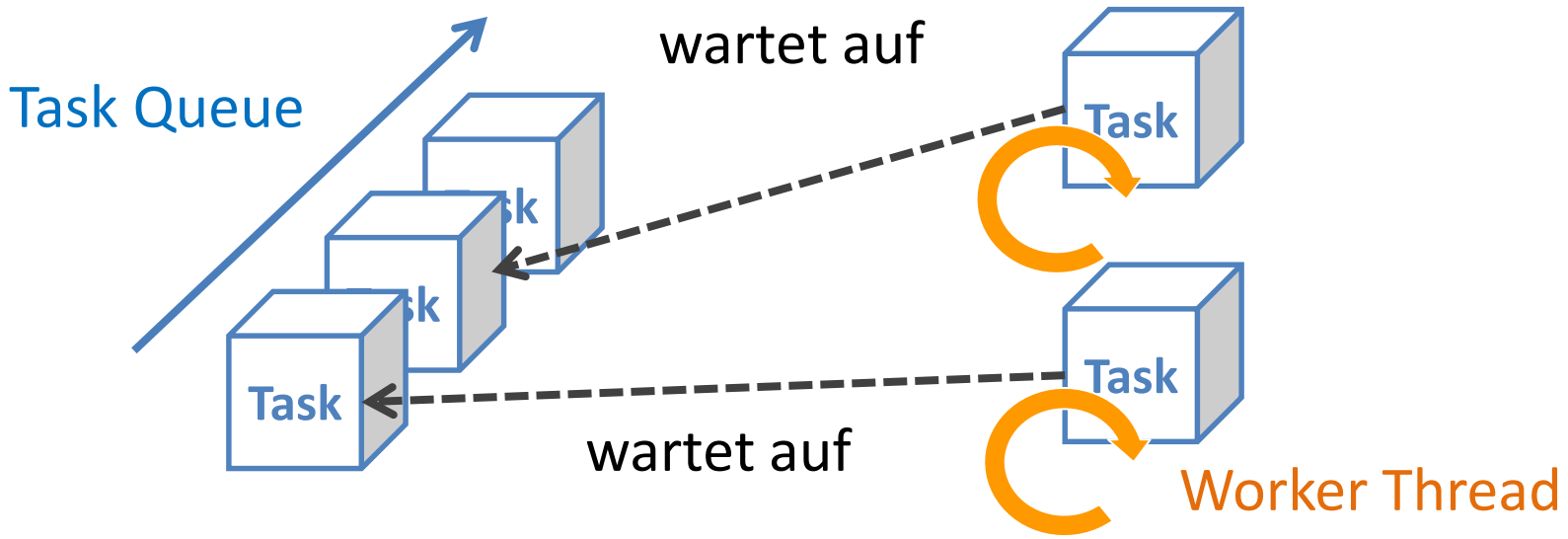
wartet auf

```
Task.Run(() => {
  ...
  signal.Set();
});
```

Skalierungsproblem

Task Warte-Abhängigkeiten

- Deadlock bei begrenzter Anzahl Worker Threads
- Thread Injection: TPL fügt neue Threads hinzu
- Geschieht langsam, viele Threads möglich

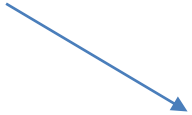


Task Continuations als Lösung: task1.ContinueWith(task2)

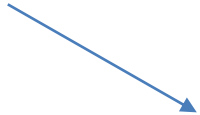
7. Uber-Asynchrony

- Grassierende Asynchronität bis in kleinste Methoden

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



```
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 (Pulse)

```
lock(this) {
  if (Full) Monitor.Wait(this);
  queue.Enqueue(x);
  Monitor.Pulse(this);
}
```

```
lock(this) {
  if (Empty) Monitor.Wait(this);
  var x = queue.Dequeue();
  Monitor.Pulse(this);
}
```

Typische Monitor-Fehler

- Wartebedingung immer wiederholt prüfen
  - `while (Full) Monitor.Wait(this);`
  - Andere Threads können vor dem aufgeweckten Thread drankommen (Signal-And-Continue)
- Bei mehreren Wartebedingungen: Signal an alle
  - `Monitor.PulseAll(this);`
  - Sonst wird evtl. nur ein Thread der falschen Bedingung geweckt (z.B. Wartender auf nicht-leer statt nicht-voll)

9. Atomic, Volatile and Yield

- Atomaren Instrukionen (Interlocked)
- Volatile Variablen, Memory Barriers
- Thread Yield, Spin-Locks

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

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

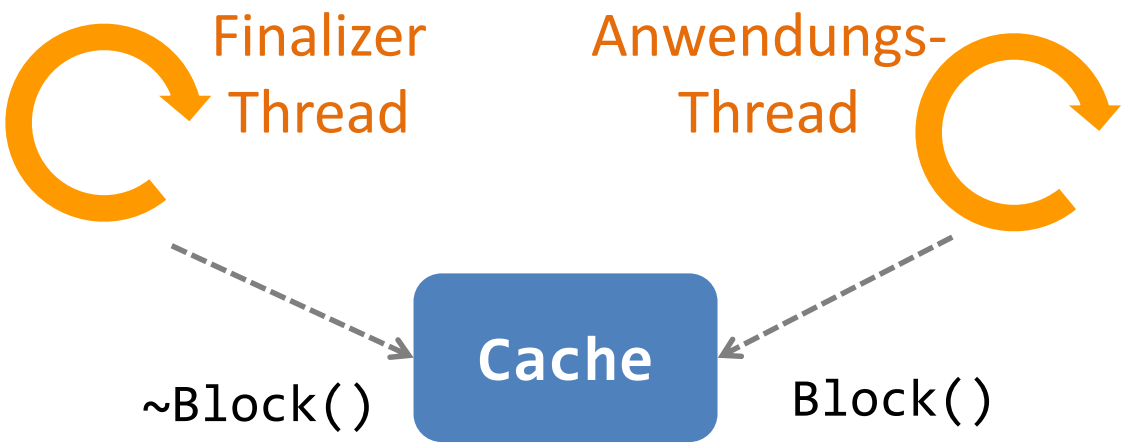
```
public class Block {
   public Block() {
      Cache.NofBlocks++;
   }

   ~Block() {
      Cache.NofBlocks--;
   }
}
```

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 .NET
  - Probleme sind allgemein, gleich z.B. in Java
- Es gibt weitere Code Smells
  - Jeder kann weitersammeln
- Kein Absolutismus
  - Nicht jeder Smell zeigt einen Fehler

Danke für Ihre Aufmerksamkeit

- Concurrency Research, Consulting, und Training
  - http://concurrency.ch
- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    IFS Institut für Software
    Rapperswil, Schweiz
  - lblaeser@hsr.ch