Parallel Code Smells: A Top 10 List

Luc Bläser

Hochschule für Technik Rapperswil

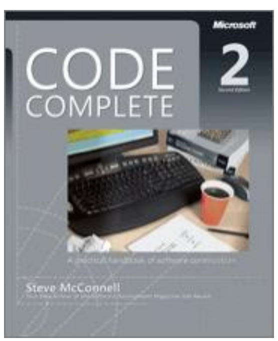
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

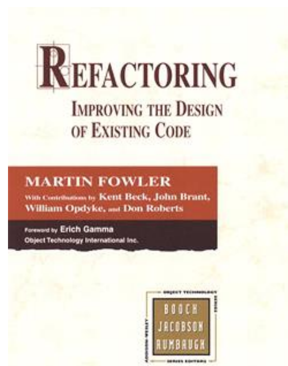
HSR Concurrency Lab
Prof. Dr. Luc Bläser

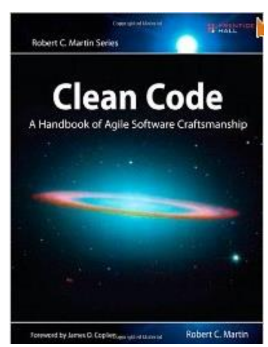
Multicore@Siemens
8 Feb. 2017, Nuremberg

Code Smells

- Symptoms in code
  - Indicators of potential design flaws
- Partly curable by refactoring
  - Restructuring without change of behavior
- Until now, focus on sequential OO
  - E.g. Huge classes, too many parameters, down casts







Parallel Code Smells

- Focus on concurrency and parallelization
  - By the example of .NET and Java
  - Also applicable for other languages
- Personal collection
  - Gained by code reviews in industry
  - Last 5 years, prioritized by relevance

The Top 10 List

Earlier presentations: OOP 2017, Parallel 2016, Heise Developer July 2016

1. Partly Synchronized Class

- Synchronized and unsynchronized externally accessible members within the same class

Java

```
class BankAccount {
   private int balance;

   public int getBalance() { return balance; }

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

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

unsynchronized

synchronized

unsynchronized

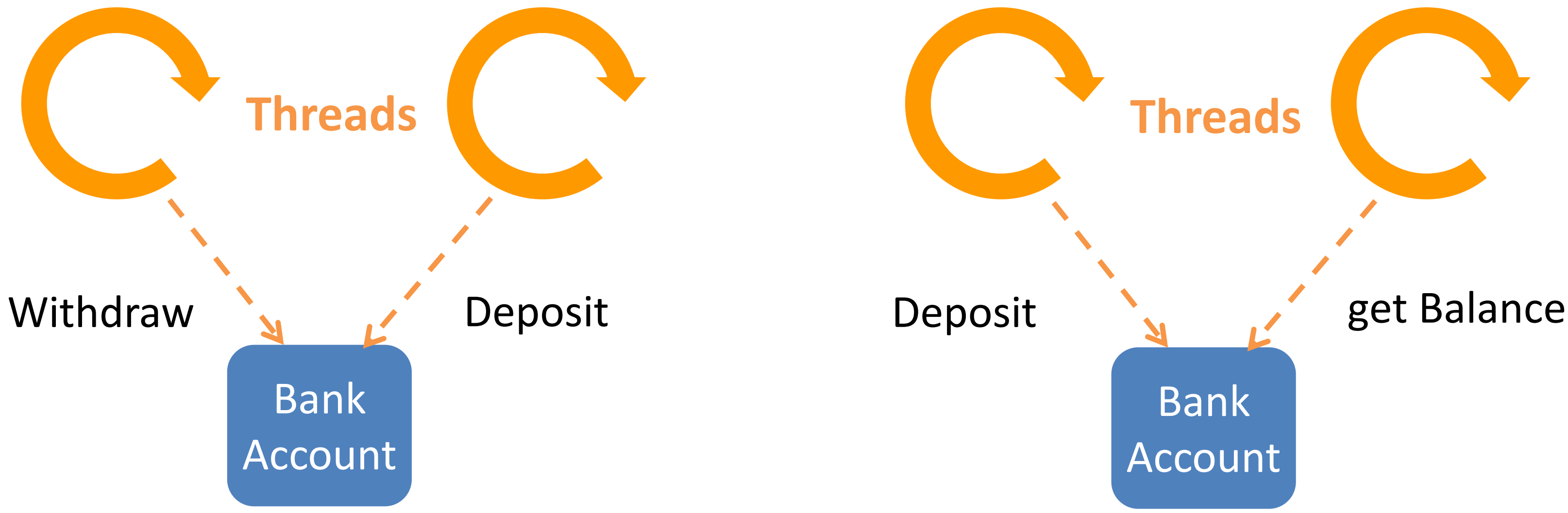
Analogous in .NET

C#

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

   public int Balance { get; private set; }        unsynchronized

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

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

Problem: Half Thread-Safe

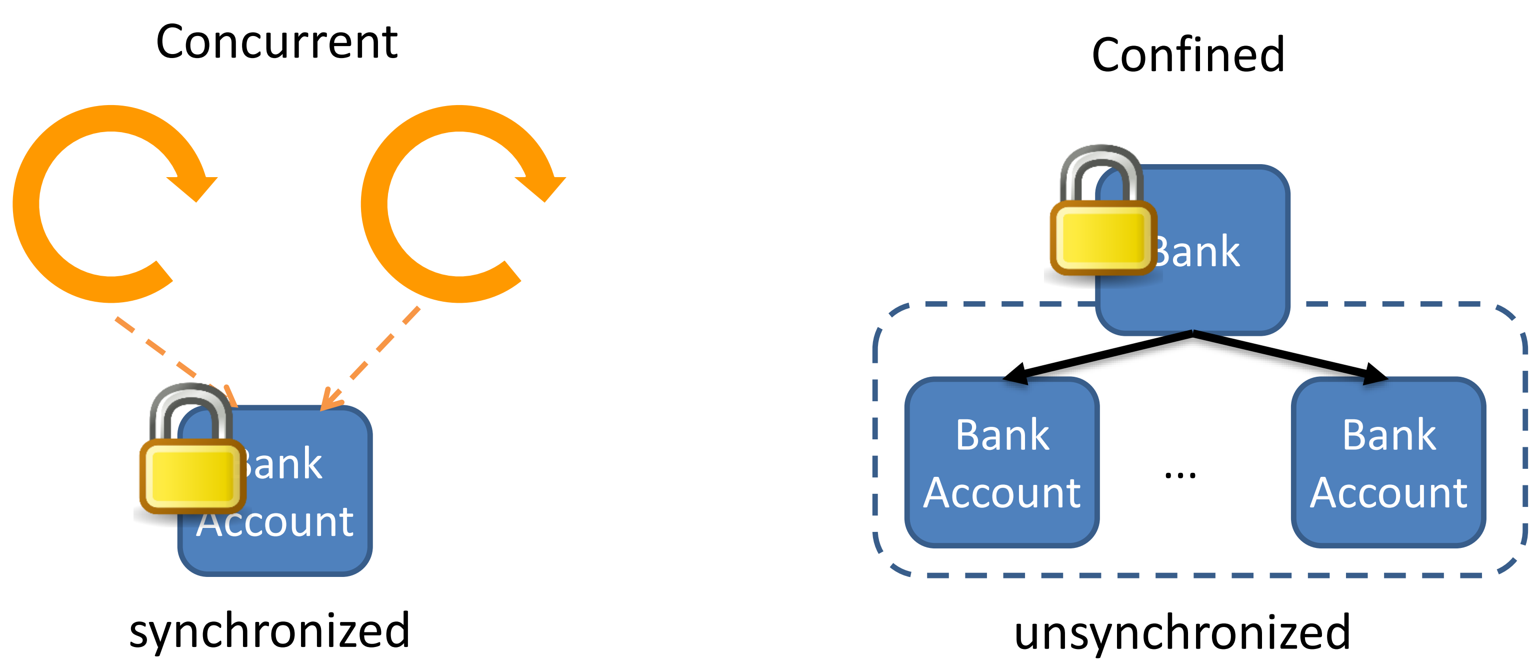
- Only concurrent Deposit/Deposit is thread-safe
- Other combinations not



Data Races & Race Conditions

Cure: Proper Architecture

- Which threads access which objects?
- Defined coherent usage per class/object



2. Nested Locking Through Method Calls

- Synchronized method directly or indirectly calls a synchronized method

Java

```
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

Hidden Nested Locks

Thread 1

lock a
lock b

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

Thread 2

lock b
lock a

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

T1 locks a
T2 locks b
T1 wants b
T2 wants a

Deadlock

Same Problem in .NET

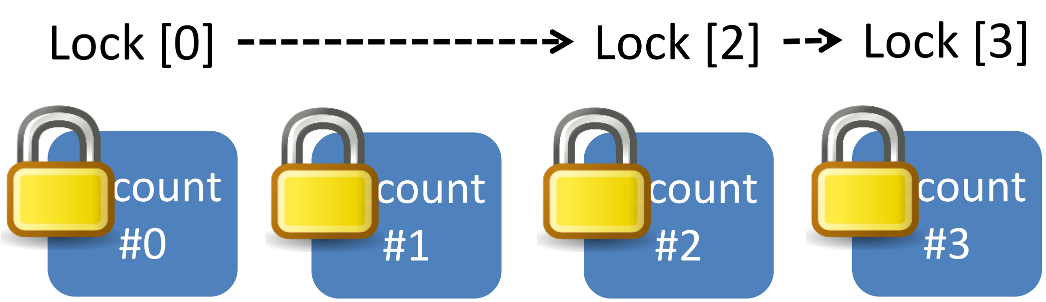
C#

```
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);
      }
   }
}
```

lock this.sync
lock target.sync

Cure: Proper Architecture

- Where are locks acquired and in which nested order?
- Avoid nested locks
- Or ensure a linear ordering



Lock the accounts only by increasing number

3. Try-and-Fail Resource Acquisition

- Repeated lock attempts without blocking or with timeouts

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

Java

Starvation

Solution: Prefer blocking synchronization primitives

4. Use of Explicit Threads

- Starting explicit threads

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

Java

Poor scalability:
=> Too many threads: out of memory

Cure: Tasks Instead of Threads

- Management in a thread pool
  - Task = potentially parallel execution
  - Limited amount of worker threads
  - Scales well, recycles threads

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

Java (Common Fork Join Pool)

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

C# (.NET TPL)

5. Thread Pool Task Dependencies

- Tasks await conditions of other tasks
  - Exception: Joining sub-tasks is okay

Java

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

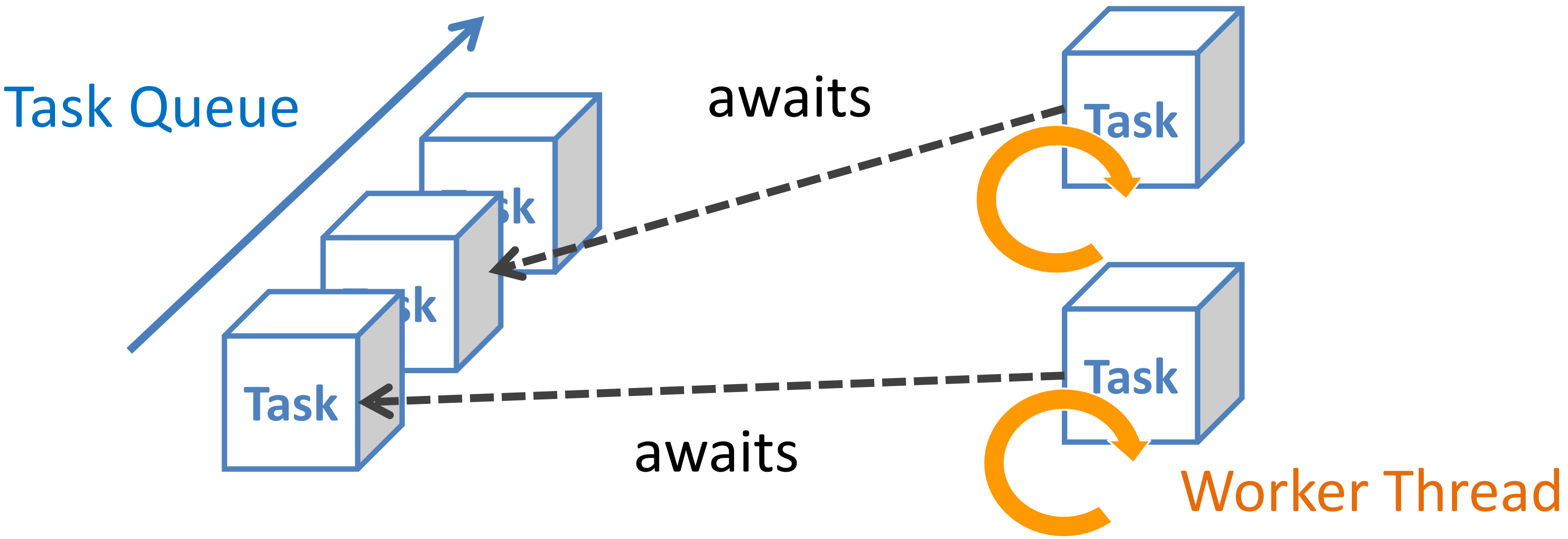
awaits

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

Deadlock or Scalability Issue

Task Wait Dependencies

- Deadlock in Java: limited amount of worker threads
- Inefficient in .NET: TPL slowly adds threads



Solution: Task continuations

6. Fire and Forget

- Launching tasks without later awaiting their end or result

Java

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

C#

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

Various issues

Problems with Fire And Forget

- Exceptions in task are ignored
  - In Java and .NET since version 4.5

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

- Application may stop before task end
  - .NET TPL and Java ForkJoinPool use daemon threads

```
CompletableFuture.runAsync(() => {
    …
                sudden end
    …
}
```

7. Uber-Asynchrony

- Rampant asynchrony down to the smallest method

```
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) {
                    ...
                  }
```

C#

Unnecessary Complexity

- Unclear, many thread switches
- Synchronous logic, run it asynchronously as a whole
  - Exception: if UI operations happen within the methods

```
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

- Wait in monitor without loop
- Single signal

Java

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

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

Common Monitor Pitfalls

- Check wait condition repeatedly
  - `while (full) wait();`
  - Other threads can overtake the signaled thread (signal and continue)
- Multiple wait conditions => signal to all
  - `notifyAll();`
  - A treads of the wrong condition may be waked up (non-empty vs. non-full)
- Same applies to .NET!

9. Atomic, Volatile, and Yield

- Atomic instructions
- Volatile variables
- Thread yield, spin locks

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

C#

Lock-Free Programming

- Complex, error-prone, often inefficient
  - Memory model expertise is mandatory
- Unnecessary in application software
  - Exception: Low-level algorithms/data structures

Read without memory fence

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

if and Add are not atomic

Wrong

10. Finalizers Accessing Shared State

- Finalizers accessing shared resources

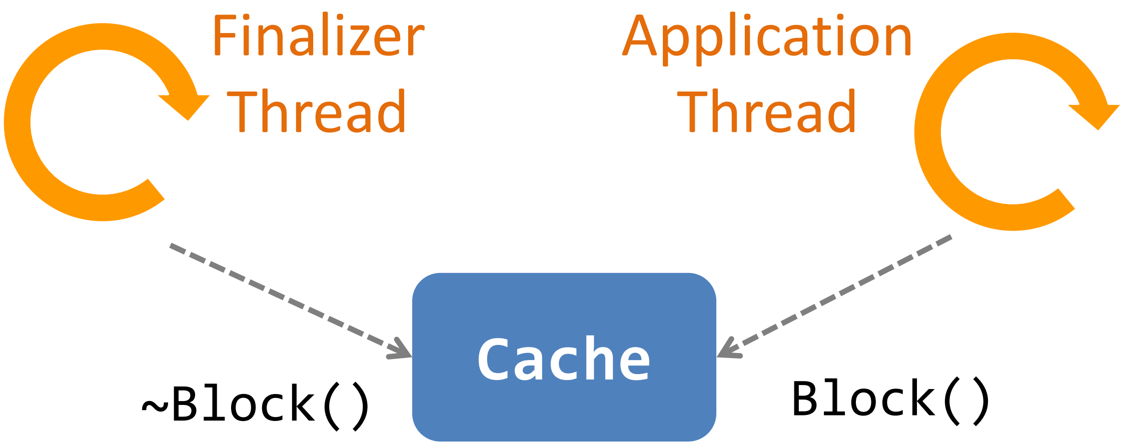
```
public class Block {
   public Block() {
      Cache.NofBlocks++;
   }

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

C#

Data Races & Race Conditions

Analysis: Finalizer

- Finalizer run concurrently to the application
- Proper synchronization is needed



Conclusions

- Code smells for parallel/concurrent aspects
  - Raising awareness for frequent design flaws
- Examples for Java and .NET
  - Generally, same problems in other languages
- There exist more code smells
  - Everyone may collect
- No absolutism
  - Not every smells denotes an error

Thank You for Your Attention

- Contact
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    lblaeser@hsr.ch
  - **HSR Concurrency Lab**
    - http://concurrency.ch
  - **Microsoft Innovation Center Rapperswil**
    - http://msic.ch

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

HSR Concurrency Lab
Prof. Dr. Luc Bläser

Microsoft
Innovation Center Rapperswil