

Strategized Locking Pattern

Pattern-Oriented Software Architecture, Vol. 2 (POSA2)

Florian Merlau
Master Informatik – Advanced Software Quality (WiSe 25/26)

27. Oktober 2025

Agenda

Einführung & Motivation

Patternbeschreibung

Codebeispiel und Diagramm

Verwandte Patterns

Bewertung & Fazit

Motivation

- ▶ Nebenläufige Software ist fehleranfällig: *Race Conditions, Deadlocks*
- ▶ Feste Locking-Strategien sind unflexibel und schwer wartbar
- ▶ Ziel: Wiederverwendbare Komponenten mit austauschbaren Synchronisationsmechanismen

Problem klassischer Ansätze

- ▶ Hard-coded Locks → geringe Flexibilität
- ▶ Mehrere Varianten desselben Codes für unterschiedliche Threading-Modelle
- ▶ Schwierige Wartung, fehleranfällig, Performance-Verlust

Intent und Kontext

Intent:

Strategize a component's synchronization to increase its flexibility and reusability without degrading performance or maintainability. (Schmidt, 1998)

Kontext:

- ▶ Wiederverwendbare Komponenten, die in verschiedenen Concurrency-Umgebungen laufen müssen
- ▶ Locking soll konfigurierbar oder „pluggable“ sein

Problem & Forces

- ▶ Unterstützung verschiedener Locking-Strategien (Mutex, RW-Lock, Null-Mutex)
- ▶ Anforderungen: Performance, Wartbarkeit, Portabilität
- ▶ Trade-off: Compile-Time Templates vs. Run-Time Polymorphie



Lösung – Strategized Locking

- ▶ Synchronisation wird ausgelagert in eine Strategie-Komponente
- ▶ Zwei Varianten:
 1. **Polymorph** – Lock-Typ wird zur Laufzeit gewählt
 2. **Template-basiert** – Lock-Typ wird beim Kompilieren festgelegt
- ▶ Nutzung des **Scoped Locking Idioms** (RAII)

Codebeispiel (C++)

```
template <class LOCK>
class File_Cache {
public:
    const char* find(const char* path) {
        Guard<LOCK> guard(lock_);
        // Zugriff auf Cache...
        return nullptr;
    }
private:
    LOCK lock_;
};
```


Diagramm – Strategized Locking

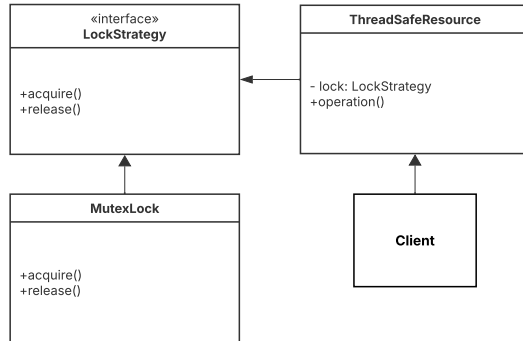


Abbildung: Eigene Darstellung des Klassendiagramms zur Strategized Locking.

Vergleich der Strategien

Strategie	Beschreibung	Vorteil	Nachteil
Null_Mutex	Kein Lock (Single-Threaded)	Schnell	Unsicher bei Threads
Thread_Mutex	Standard-Lock	Robust	Langsam bei hoher Parallelität
RW_Lock	Paralleles Lesen, exklusives Schreiben	Effizient bei vielen Reads	Komplexer



Thread-safe Interface Pattern

- ▶ Verhindert *Self-Deadlocks* durch klare Trennung
- ▶ **Interface methods check** – öffentliche Methoden sichern Zugriff
- ▶ **Implementation methods trust** – interne Methoden vertrauen auf bestehenden Lock

Scoped Locking Idiom

- RAIL-Prinzip: Lock wird beim Eintritt erworben, beim Verlassen freigegeben

```
template<class LOCK>
class Guard {
public:
    Guard(LOCK &l): lock(l) { lock.acquire(); }
    ~Guard() { lock.release(); }
private:
    LOCK &lock;
};
```



Vor- und Nachteile

Vorteile:

- ▶ Hohe Flexibilität und Wiederverwendbarkeit
- ▶ Wartungsfreundlich (eine zentrale Implementierung)
- ▶ Einfache Performance-Anpassung

Nachteile:

- ▶ Template-Komplexität bzw. Sichtbarkeit der Strategien
- ▶ Potenziell höhere Kompilierzeit

Bekannte Anwendungen

- ▶ **ACE Framework (Adaptive Communication Environment)**
- ▶ **Booch Components**
- ▶ Moderne Äquivalente: `std::lock_guard`, Java `synchronized`

Fazit

- ▶ Strategized Locking = “pluggable synchronization”
- ▶ Klare Trennung von Funktionalität und Synchronisation
- ▶ Grundlage vieler moderner Concurrency-Patterns





Diskussion

- ▶ Wann lohnt sich Strategized Locking in modernen Systemen?
- ▶ Welche Alternativen gibt es in Java, C++, Rust?



Quellen I

-  Douglas C. Schmidt, *Strategized Locking, Thread-safe Interface, and Scoped Locking*, C++ Report, Washington University, 1998.
-  Buschmann, F., Schmidt, D. C., et al. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.