CLEAN ARCHITECTURE

Maurice Müller

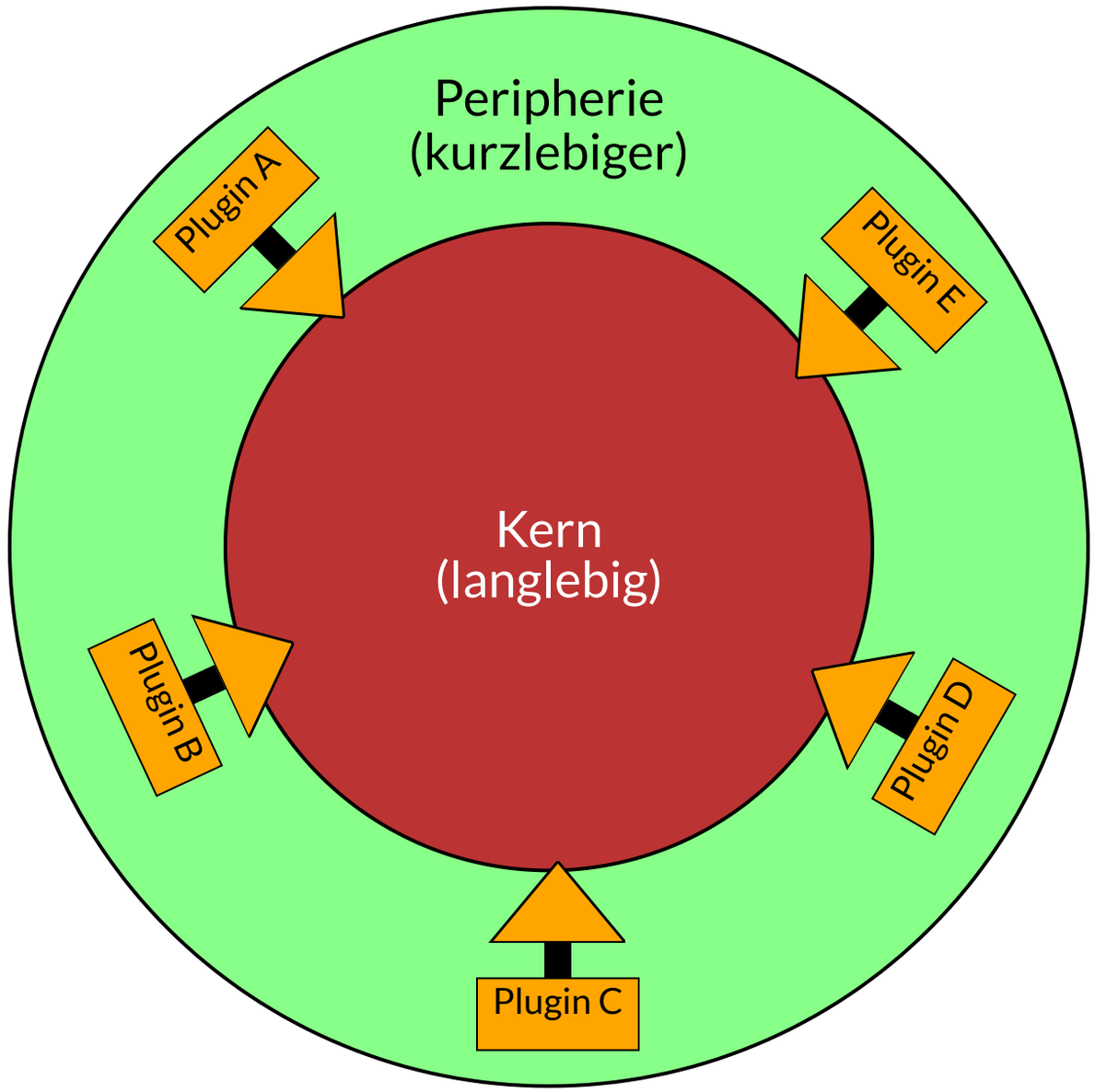
2023-12-04

Clean Architecture

Nachhaltige Architektur

- eine langfristige Architektur
  - besitzt einen technologieunabhängigen Kern (= die eigentliche Anwendung)
  - behandelt jede Abhängigkeit als temporäre Lösung
  - unterscheidet zwischen zentralem (langlebigem) und peripherem (kurzlebigerem) Code
- Metapher: die Zwiebel
  - *Onion Architecture*

Struktur der Clean Architecture



- **Abhängigkeiten immer von außen nach innen**
- Kern-Code hängt nie von Plugins ab

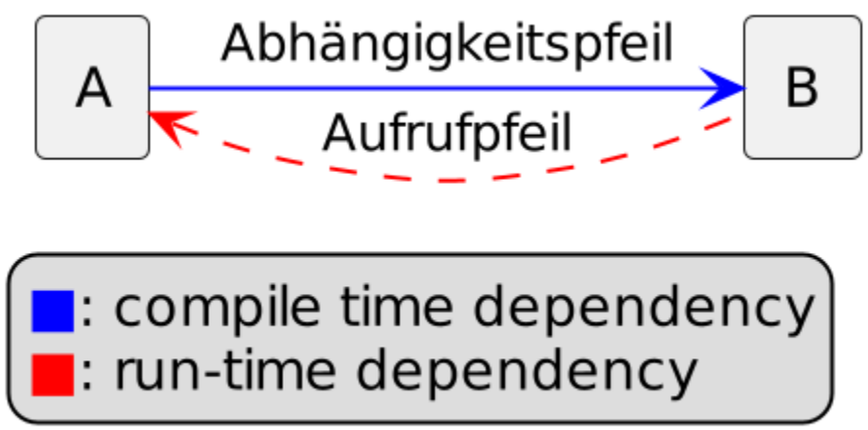
Dependency Rule

- zentrale Regel für Abhängigkeiten

Abhängigkeiten immer von außen nach innen

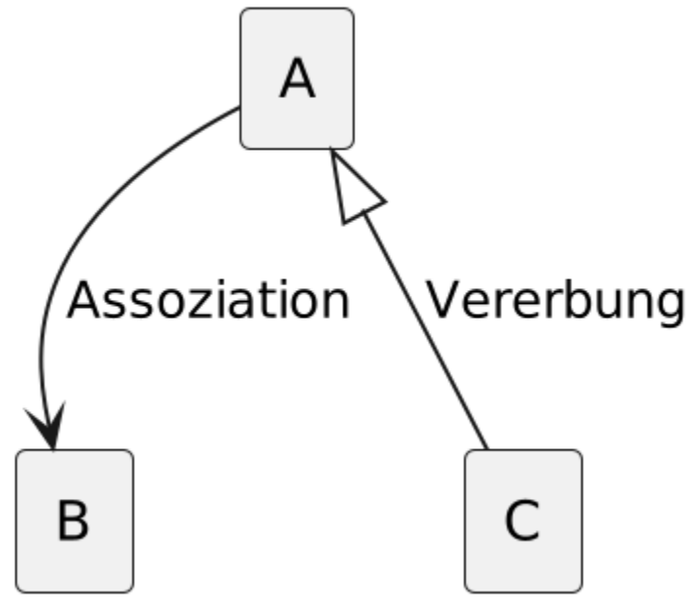
- erfordert für jede Klasse eine klare Positionierung
- Abhängigkeitspfeile gehen immer von außen nach innen
  - Aufrufpfeile können in beide Richtungen gehen

Bedeutung der Pfeile



- Abhängigkeitspfeil: Code A benötigt Code B zum Kompilieren
  - Code A referenziert Code B direkt mit Namen
- Code B ruft während der Ausführung Elemente von Code A auf
  - Code B bekommt erst zur Laufzeit eine Referenz auf Code A

Zwei Arten von Abhängigkeiten



- Assoziation: Code A hält eine Referenz auf Code B (und verwendet diese)
- Vererbung: Code C übernimmt das Verhalten von Code A und variiert es an passenden Stellen

Abhängigkeiten gestalten

- Software-Architektur ist die Kunst, die Abhängigkeiten zwischen Systemteilen willentlich und zum Vorteil der Beteiligten zu gestalten
- die Richtung und Art der Pfeile im Architektur-diagramm festzulegen, ist die Aufgabe des Software-Architekten
- die Richtung kann beliebig gewählt werden
- wir können die Richtung jederzeit umdrehen!

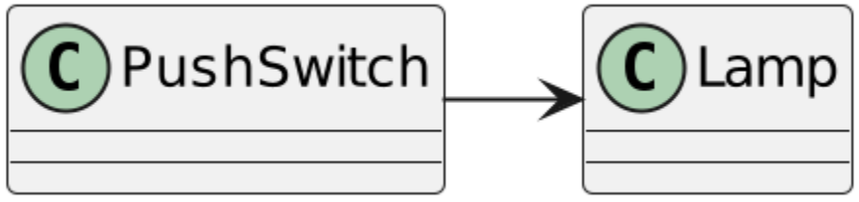
Abhängigkeiten: Beispiel

```java
public class PushSwitch {
    private boolean pushed = false;
    private final Lamp lamp;

    public PushSwitch() {
        this.lamp = new Lamp();
    }

    void push() {
        if(!pushed) {
            lamp.turnOn();
            pushed = true;
            return;
        }
        lamp.turnOff();
        pushed = false;
    }
}
```

```java
public class Lamp {
    private boolean glowing = false

    public void turnOn() {
        glowing = true;
    }

    public void turnOff() {
        glowing = false;
    }
}
```

Wer hängt von wem ab und wann?

Abhängigkeiten: Beispiel



PushSwitch hängt zur Compilezeit von *Lamp* ab

Abhängigkeiten: Beispiel

Verbesserung: Dependency Injection

```java
public class PushSwitch {
    private boolean pushed = false;
    private final Lamp lamp;

    public PushSwitch(Lamp lamp) {
        this.lamp = lamp;
    }

    void push() {
        if(!pushed) {
            lamp.turnOn();
            pushed = true;
            return;
        }
        lamp.turnOff();
        pushed = false;
    }
}
```

```java
public class Lamp {
    private boolean glowing = false

    public void turnOn() {
        glowing = true;
    }

    public void turnOff() {
        glowing = false;
    }
}
```

Wer hängt von wem ab und wann?

Es hat sich nichts geändert.

Abhängigkeiten: Beispiel

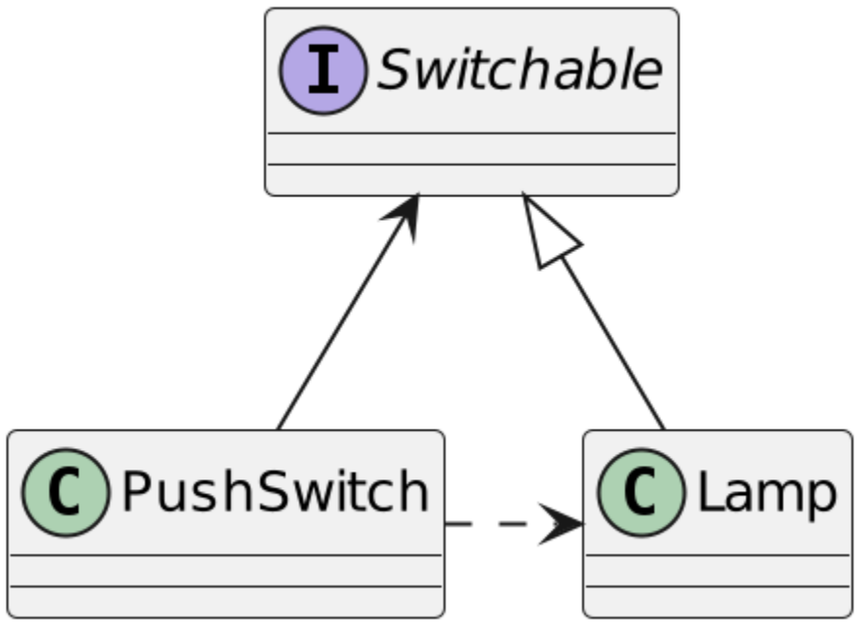
Verbesserung: Dependency Inversion

```
public class PushSwitch {
    private boolean pushed = fal
    private final Lamp lamp;

    public PushSwitch(Lamp lamp)
        this.lamp = lamp;
    }

    void push() {
        if(!pushed) {
            lamp.turnOn();
            pushed = true;
            return;
        }
        lamp.turnOff();
        pushed = false;
    }
}
```

```
public interface Switchable {
    public void turnOn();
    public void turnOff();
}

public class Lamp implements  Switchab
    private boolean glowing = false;

    public void turnOn() {
        glowing = true;
    }

    public void turnOff() {
        glowing = false;
    }
}
```

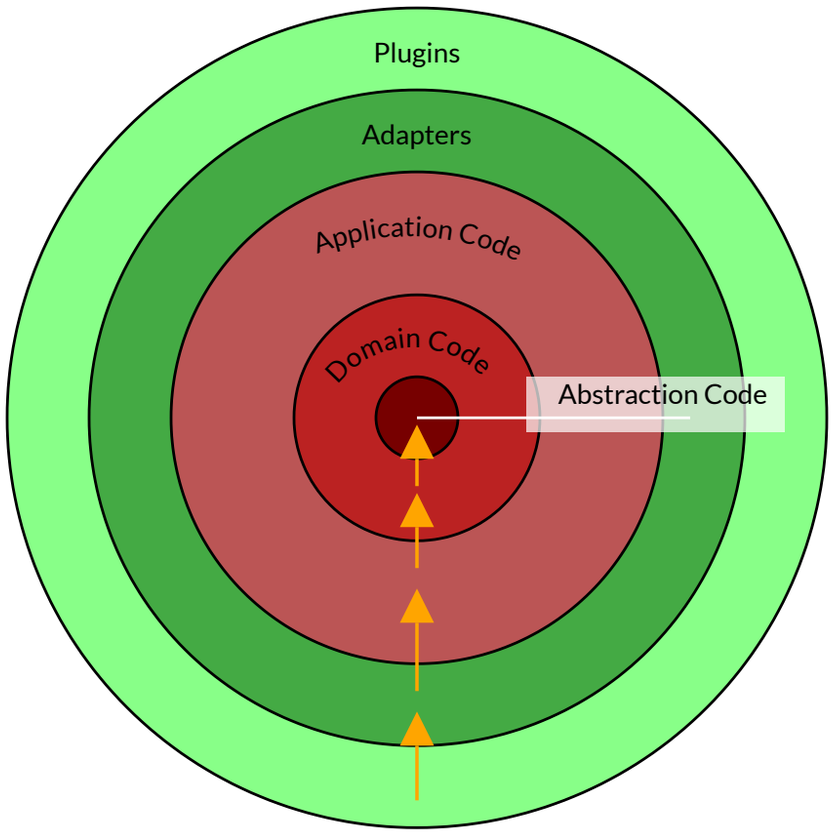
Wer hängt von wem ab und wann?

Abhängigkeiten: Beispiel



- *PushSwitch* hängt zur Compilezeit von *Switchable* ab
- *PushSwitch* hängt zur Laufzeit von *Lamp* ab
- *Lamp* hängt zur Compilezeit von *Switchable* ab

Struktur der Clean Architecture



- innere Schichten wissen nichts von den Äußeren
  - Abhängigkeiten immer von außen nach innen
- beliebig viele innere Schichten (oft drei)

Grundregeln der Clean Architecture

- Anwendungs- und Domaincode ist frei von Abhängigkeiten
  - sämtlicher Code kann eigenständig verändert werden
  - sämtlicher Code kann unabhängig von Infrastruktur kompiliert und ausgeführt werden
- innere Schichten definieren Interfaces, äußere Schichten implementieren diese
- äußere Schichten koppeln sich an die inneren Schichten

Abstraction Code

- enthält domänenübergreifendes Wissen
  - mathematische Konzepte (z.B. Matrizen)
  - Algorithmen und Datenstrukturen
  - abstrahiere Muster (z.B. *Language*)
- häufig nicht notwendig und/oder nicht vorhanden
- wahrscheinlich bereits als Library verfügbar
- kann nachträglich extrahiert werden

Domain Code

- enthält v.a. Entities (Business Objects)
- implementiert organisationsweit gültige Geschäftslogik (Enterprise Business Rules)
- der innere Kern der Anwendung bzw. Domäne
- sollte sich am seltensten ändern
  - immun gegen Änderungen an Details wie Anzeige, Transport oder Speicherung
  - unabhängig vom konkreten Betrieb der Anwendung
- hier sollte viel Gehirnschmalz reinfließen

Application Code

- enthält die Anwendungsfälle (Use Cases)
  - resultiert direkt aus den Anforderungen
- implementiert die anwendungsspezifische Geschäftslogik (Application-specific Business Rules)
- steuert den Fluss der Daten und Aktionen von und zu den Entities
  - verwendet die Geschäftslogik, um den jeweiligen Anwendungsfall umzusetzen

Application Code

- Änderungen in dieser Schicht beeinflussen die nicht die weiter inner liegenden Schichten
- isoliert von Änderungen an der Datenbank, der GUI, HTTP-API, etc.
- wenn sich Anforderungen ändern, hat das wahrscheinlich Auswirkungen auf diese Schicht
- wenn sich der konkrete Betrieb der Anwendung ändert, kann das hier Auswirkungen haben

Adapters

- vermittelt Aufrufe von Daten an die inneren Schichten
  - Formatkonvertierungen
    - externes Format wird umgewandelt, damit die Applikation es verarbeiten kann
    - internes Format wird umgewandelt, damit externe Plugins es verarbeiten können
- oftmals nur einfache Datenstrukturen, die hin- und hergereicht werden
- Ziel: Entkopplung von *innen* und *außen*
- Anti-Corruption Layer

Plugins

- diese Schicht greift grundsätzlich nur auf die Adapter zu
- enthält Frameworks, Datentransportmittel und andere Werkzeuge
  - v.a. Datenbanken, GUIs, Web
  - alle *Pure Fabrication*-Entscheidungen
- hier sollte möglichst wenig Code geschrieben werden
  - hauptsächlich Delegationscode, der an die Adapter weiterleitet

Plugins

- auf gar keinen Fall enthält diese Schicht Anwendungslogik
  - Daten fallen mundfertig aus dem Adapter
  - alle Entscheidungen sind bereits gefallen
  - Anfragen werde nicht uminterpretiert (das wäre Aufgabe der Adapter)
- keine emotionale Bindung an diesen Code
  - jederzeitige Änderungen möglich
  - Auswirkungen nur auf die Adpaterschicht
  - übersichtlicher Aufwand

Ziel der Clean Architecture

- das Ziel ist, Code nur noch von langlebigerem Code abhängig zu machen
- wenn sich Technologien ändern müssen, kann die Anwendung unverändert bleiben
- Grenzen
  - technische Grundlagen müssen stabil bleiben
    - Plattform SDK, Programmiersprache/Syntax, Compiler, Laufzeitumgebung
  - auch Betriebssystem und Hardware benötigen Stabilität