Design Pattern

	Klassenbasiert	Objektbasiert
Creational Patterns (Erzeugungsmuster)	Factory Method	Abstract Factory Builder Prototype Singleton
Structural Patterns (Strukturmuster)	Adapter	Adapter Bridge Composite Decorator (Delegation) Facade Flyweight Proxy
Behavioral Patterns (Verhaltensmuster)	Interpreter Template Method	Chain of Responsibility Command Iterator Mediator **Memento** Observer(Listener/Model-View-Controller) State Strategy (Hook) Visitor

Design Pattern: Memento

- Auch: *Token*
- Zweck:
  - Speichern des internen Zustands eines Objekts ermöglichen
  - Wiederherstellen eines Objekts in einem vorherigen Zustand ermöglichen
- Bietet Kapselung, indem Details des gespeicherten Zustands verborgen bleiben
  - Kapselung: Attribute sollen nicht außen bekannt sein; stattdessen lediglich schmale Schnittstelle
- `https://en.wikipedia.org/wiki/Memento_pattern`
- `https://refactoring.guru/design-patterns/memento`

Memento Pattern

- Das Memento Pattern verwendet drei Hauptkomponenten:
  - `Originator`: Das Objekt, dessen Zustand gespeichert werden soll
  - `Memento`: Ein Container, der den gespeicherten Zustand enthält
  - `Caretaker`: Verwaltet die `Mementos` und stellt den Zustand des `Originators` wieder her
- Der Zustand wird im `Memento`-Objekt gekapselt, um die Kapselung des `Originators` zu erhalten

Memento Pattern—Code

```
class Originator { // Objekt, dessen Zustand gespeichert werden soll
  private String state;
  public void setState(String state) { this.state = state; }
  public Memento saveState() { return new Memento(state); }
  public void restoreState(Memento m) { this.state = m.getState(); }
}

class Memento { // Container, der den gespeicherten Zustand enthaelt
  private String state;
  public Memento(String state) { this.state = state; }
  public String getState() { return state; }
}

class Caretaker { // Memento-Verwaltung und Originator-Wiederherstellung
  private List<Memento> mementos = new ArrayList<>();
  public void addMemento(Memento m) { mementos.add(m); }
  public Memento getMemento(int index) { return mementos.get(index); }
}
```

Memento Pattern—Livedemo

- **Speichern von Autos**

Memento Pattern—Livedemo

- **Speichern von Autos**
- **Positiv:**
  - **Kapselung: Attribute von `Originator` und `Memento` können *nicht* gelesen werden**
- **Negativ:**
  - **`Auto` ist selbst für `Memento` verantwortlich—keine saubere Trennung**
  - **Problem des “Klonen” tritt hier wieder auf (vgl. Prototype Pattern)**

Memento Pattern

- **Ziel: Zustand eines Objekts extern speichern**
  - **Dabei muss auf die Attribute des Objekts zugegriffen werden**
- **Die Kapselung soll möglichst nicht geschwächt werden**
- **Umsetzung: `Originator` erstellt ein `Memento`**
  - **`Caretaker` speichert die `Mementos`**

Memento Pattern

GoF: *“Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.”*

Design Pattern

	Klassenbasiert	Objektbasiert
Creational Patterns (Erzeugungsmuster)	Factory Method	Abstract Factory Builder Prototype Singleton
Structural Patterns (Strukturmuster)	Adapter	Adapter Bridge Composite Decorator (Delegation) Facade Flyweight Proxy
Behavioral Patterns (Verhaltensmuster)	Interpreter Template Method	Chain of Responsibility Command Iterator Mediator Memento Observer(Listener/Model-View-Controller) State Strategy (Hook) Visitor

Wiederholung: Ziele von Design Pattern

- **Wiederverwendbarkeit**
  - Code soll in verschiedenen Kontexten wiederverwendet werden können
- **Wartbarkeit und Verständlichkeit**
  - Systeme sollen leicht wartbar und verständlich sein
- **Flexibilität**
  - Systeme sollen sich leicht an neue Anforderungen anpassen lassen
- **Entkopplungen**
  - Abhängigkeiten zwischen Komponenten sollen minimiert werden, damit Änderungen in einer Komponente nicht andere negativ beeinflussen
- **Komplexität**
  - Komplexität soll durch bewährte Lösungen für wiederkehrende Probleme reduziert werden

Wiederholung: Ziele von Design Pattern

- **Wiederverwendbarkeit**
  - Code soll in verschiedenen Kontexten wiederverwendet werden können
- **Wartbarkeit und Verständlichkeit**
  - Systeme sollen leicht wartbar und verständlich sein
- **Flexibilität**
  - Systeme sollen sich leicht an neue Anforderungen anpassen lassen
- **Entkopplungen**
  - Abhängigkeiten zwischen Komponenten sollen minimiert werden, damit Änderungen in einer Komponente nicht andere negativ beeinflussen
- **Komplexität**
  - Komplexität soll durch bewährte Lösungen für wiederkehrende Probleme reduziert werden

→ **Alle diese Ziele können noch weiter unterstützt werden!**

Was ist Virtualisierung?

- *Virtualisierung* ist ein *"Nachbildung eines Hard- oder Software-Objekts durch ein ähnliches Objekt vom selben Typ mit Hilfe einer Abstraktionsschicht"*
- Die Erzeugung einer virtuellen Version einer Ressource, z.B.
  - eines Betriebssystems,
  - eines Speichers,
  - einer Netzwerkressource,
  - etc.

Was ist Virtualisierung?

- Virtualisierung ermöglicht das gleichzeitige Ausführen mehrerer Betriebssysteme auf demselben physischen Computer
- Diese mehreren Ausführungen sind voneinander isoliert
- Durch Virtualisierung können Hardware-Ressourcen effizienter genutzt werden

Arten der Virtualisierung

- **Betriebssystemvirtualisierung**
  - **Mehrere Betriebssysteminstanzen laufen auf einem einzigen physischen Host**
- **Hardwarevirtualisierung (Vollvirtualisierung)**
  - **Ein ganzes Computer-System wird simuliert**
- **Paravirtualisierung**
  - **Das Gast-Betriebssystem ist sich seiner Virtualisierung bewusst und interagiert direkt mit der Virtualisierungssoftware**

Virtualisierungs-Software

- **Microsoft Hyper-V Manager**
- **VMware Workstation**
- **Oracle VirtualBox**
- **Parallels Desktop for Mac**
- **…**
- ***Docker***
- ***Podman***

Was ist Docker?

- Gestartet im März 2013
- Open-Source
  - `https://www.docker.com/community/open-source`
- Aktuelle Version, Januar 2025: 27.5.0
- Problem: *Works on my machine!*
- Lösung mit Docker: *Build once... (finally) run anywhere*
- $\rightarrow$ Docker automatisiert die Bereitstellung, Skalierung und Verwaltung von Anwendungen durch Containerisierung.
  - Genau genommen *nicht* durch Virtualisierung!

Was ist Docker?

- Sehr weit verbreitet
  - Quasi-Standard in der Industrie, besonders Cloud
- Immer mehr Alternativen in letzter Zeit
  - Gleiches Containerformat
  - z.B. Podman (ohne Daemon)

- Docker implementiert eine High-Level-API, um leichtgewichtige Container bereitzustellen, die Prozesse isoliert ausführen
- Ein physischer Host kann viele Docker-Container ausführen

Was ist Docker?

- Ein Docker-*Container* ist
  - leichtgewichtig
  - eigenständig
  - ausführbar
- Ein Docker-*Container* enthält alls, was benötigt wird, um eine Anwendung auszuführen
  - Code
  - Laufzeitumgebung
  - Systemwerkzeuge
  - Systembibliotheken
  - Einstellungen

Container vs. Virtualisierung

- *Containering* ist eine Methode, um mehrere Instanzen eines Betriebssystems (als "Gäste") isoliert voneinander den Kernel eines Hostsystems nutzen zu lassen
- Containering ist deutlich ressourcenschonender als eine volle Virtualisierung
- Container sind nicht so sicher wie virtuelle Maschinen
- Container speichern Daten nicht dauerhaft
- Containering ist nativ lediglich auf Linux verfügbar
  - z.B.: unter Windows und Mac werden Container innerhalb von virtuellen Maschinen ausgeführt
  - Ab Windows 11 kann Docker das Windows-Subsystem für Linux Version 2 (WSL 2) nutzen

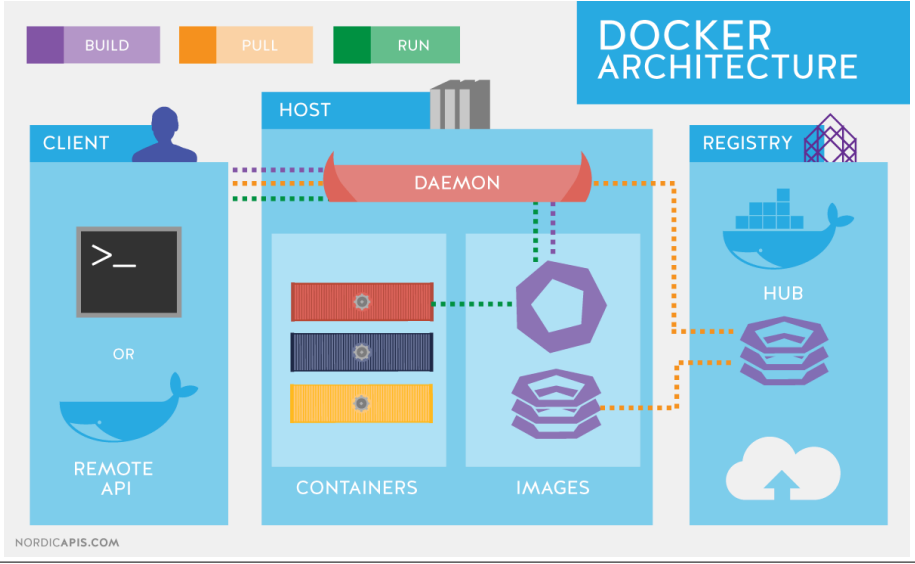
Literatur

- **Docker-Webseite**
  - `https://www.docker.com`
- **Docker-Hub**
  - `https://hub.docker.com`
- **Docker-Dokumentation**
  - `https://docs.docker.com`
- **Third-Party Tutorials**
  - `https://pointful.github.io/docker-intro`
  - `https://www.slideshare.net/Docker/docker-101-introduction-to-docker`

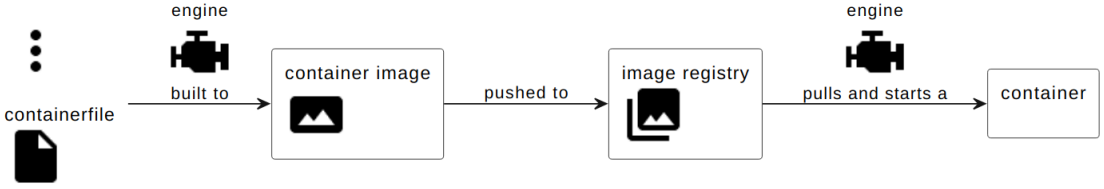
Vorteile von Docker

- **Portabilität:**
  - **Docker-Container können auf jedem System ausgeführt werden, das Docker unterstützt, unabhängig von der Umgebung**
- **Isolation:**
  - **Jeder Container ist von den anderen isoliert**
- **Skalierbarkeit:**
  - **Docker ermöglicht das einfache Hoch- und Runterskalieren von Anwendungen**
- **Versionierung:**
  - **Docker hat eingebaute Mechanismen zur Versionierung und Komponentenwiederverwendung**

Docker-Architektur



Docker-Workflow



Images und Container

- **Ein Docker-*Image* ist eine unveränderbare Datei, die die Anweisungen zum Erstellen eines Docker-*Container* enthält**
- **Docker-Images können auf Docker-Hub gespeichert und geteilt werden**
- **Docker-Images werden verwendet, um Anwendungen zu speichern und zu auszuliefern**
- **Ein Docker-*Container* ist eine laufende Instanz eines Docker-Images**

Docker Tools (Kurzausblick)

- **Docker Volume**
  - **Teilen von Daten zwischen Containern und Host**
  - **Persistieren von Daten von Containern**
- **Docker Compose**
  - **Verwaltung von mehrteilige Docker-Anwendungen**
  - **Konfiguration über YAML-Datei**
  - **Mit einem einzigen Befehl können alle Anwendungen erstellt und gestartet werden**

Docker Tools (Kurzausblick)

- **Docker Swarm**
  - **Orchestrierungstool**
  - **Verwendung von mehreren Hosts für Docker-Container**
  - **Diese Hosts werden in einem Swarm-Cluster zusammengefasst**
  - **Grundlegende Funktionen wie Skalierung und Load-Balancing**

Docker Befehle

- `docker build`
  - **Baut ein Image aus einem Docker-File**
- `docker images`
  - **Listet alle Docker-Images auf dem Host auf**
- `docker run`
  - **Führt ein Image aus**
- `docker ps`
  - **Listet alle Docker-Container auf, laufende sowie gestoppte**

Docker Befehle

- `docker stop`
  - **Stoppt ein Docker-Container**
- `docker rm`
  - **Löscht einen Docker-Container**
- `docker rmi`
  - **Löscht ein Docker-Image**

Docker—Hello World

Dockerfile:

```
# Offizielles Python-Image
FROM python:3.13-slim

# Arbeitsverzeichnis im Container
WORKDIR /usr/src/app

# Einfaches Python-Skript
RUN echo 'print("Hello, World!")' > hello.py

# Beim Start des Containers ausfuehren:
CMD ["python", "./hello.py"]
```

- **Image erstellen (im Ordner):**
  - `docker build -t hello-world .`
- **Container ausführen:**
  - `docker run hello-world`

Docker—Ziele von Design Pattern

- **Wiederverwendbarkeit**
  - **Docker-Images fördern Wiederverwendbarkeit, da sie eine standardisierte Umgebung schaffen**
  - **Ein Image kann mehrfach und in unterschiedlichen Szenarien genutzt werden, z.B. für Entwicklung, Test und Produktion**
- **Wartbarkeit und Verständlichkeit**
  - **Durch die Isolation von Anwendungen in Containern wird die Wartbarkeit verbessert**
  - **Jede Anwendung läuft unabhängig in einer eigenen, definierten Umgebung, was die Fehlerbehebung und Aktualisierung vereinfacht**

Docker—Ziele von Design Pattern

- **Flexibilität**
  - **Docker ermöglicht Flexibilität durch Container-Orchestrierung**
  - **Anwendungen können einfach skaliert, geupdatet oder in neue Umgebungen migriert werden, ohne dass die zugrunde liegende Infrastruktur geändert werden muss**
- **Entkopplungen**
  - **Container sind vollständig voneinander entkoppelt**
  - **Jede Anwendung bringt ihre eigene Laufzeitumgebung mit (inklusive Abhängigkeiten wie Bibliotheken), wodurch Konflikte zwischen Anwendungen aufgelöst werden**

Docker—Ziele von Design Pattern

- **Komplexität**
  - **Docker abstrahiert komplexe Aufgaben wie das Setup von Laufzeitumgebungen**
  - **Entwickler können sich auf die Implementierung der Anwendung konzentrieren, anstatt sich mit der Infrastruktur auseinanderzusetzen**

Docker Tutorials (Third Party)

- **What is a Container?**
  - https://turing85.github.io/articles/2023/03-23-what-are-containers/index.html
- **Containerfiles**
  - https://turing85.github.io/articles/2023/03-26-containerfiles/index.html
- **Running Containers**
  - https://turing85.github.io/articles/2023/03-29-running-containers/index.html