Implementierung

- Muster in der Softwareentwicklung
- Entwurfsmuster
- Einstieg ins Testen
- Funktionsprüfung
- Nicht funktionale Prüfung (oo):

Muster in der Softwareentwicklung

> **Muster** – Beschreibung eines wiederkehrenden Problems sowie einer bewährte und generische Lösung dafür

→ Müssen an das jeweilige Projekt angepasst werden
→ Dienen der Kommunikation von zwischen Entwicklern

> **Microservices** – Architekturstil, bei Anwendung in Dienste aufgeteilt wird, die unabhängig entwickelt / installiert werden

→ Laufen als eigenständige Prozesse, die separat bereitgestellt und skaliert werden können
→ Unterliegen einer minimalen zentralisierten Verwaltung

Entwurfsmuster

> Entwurfsmuster (design pattern) – Lösung für bestimmtes, in Zusammenhängen wiederkehrendes Entwurfsproblems

→ Faustregel: mindestens dreimal vor der Dokumentation sinnvoll eingesetzt und/oder beobachtet haben

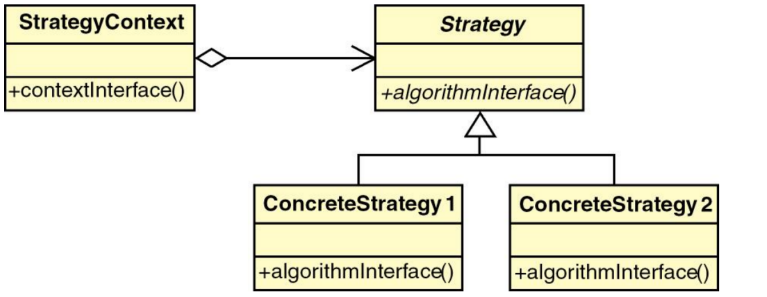
Klassifizierung von Designmustern
Creational Patterns: Helfen, *System unabhängig* davon zu erstellen, wie *Objekte* erstellt, komponiert und dargestellt werden.
Structural Patterns: *Umgang* mit der *Zusammensetzung* von Klassen und Objekten in größere Strukturen
Behavioural Patterns: Umgang mit der *Wechselwirkung* zwischen Objekten und Klassen, *komplexe Kontrollflüsse*

Beispiel: Strategy-Muster
Problem: *Verwandte Klassen unterscheiden sich dadurch, dass sie gleiche Aufgaben durch verschiedene Algorithmen lösen*
Lösung:

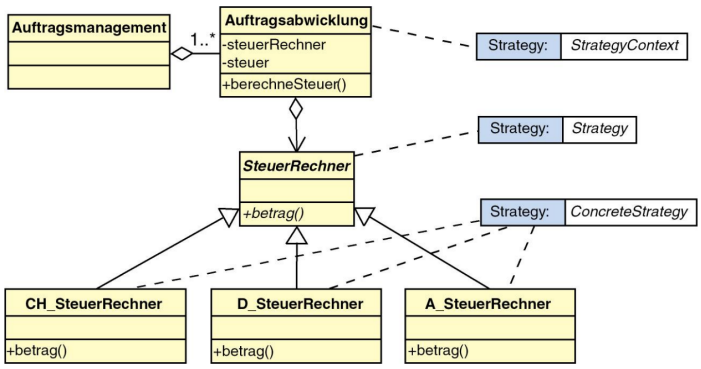
1. Eine *Klasse* definiert alle gemeinsame Operationen (**Strategy Context**)
2. Signaturen der unterschiedlich zu *implementierenden Operationen* werden in einer weiteren *Klasse* (**Strategy**)
3. Von Klasse/Schnittstelle wird für Implementierungsalternative eine konkrete Unterklasse abgeleitet (**Concrete Strategy**)
4. **Strategy Context** benutzt konkrete *Strategy-Objekte*, um unterschiedlich Operationen per Delegation auszuführen

Abstrakte Lösung:

```
Strategy strat = new ConcreteStrategy1;
```



Konkrete Lösung:



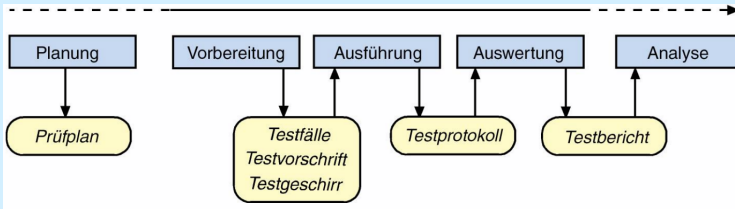
Vorteil des Strategy-Musters	Nachteile des Strategy-Musters
• Einkapselung von Algorithmen in Strategieklassen **vermeiden Fallunterschiede** zum Auswählen der jeweiligen Variante eines Algorithmus. • **Auswahl der Varianten** geschieht durch *Konfiguration des Kontextes* mit einem geeigneten Strategieobjekt.	• Erhöhter Wissensbedarf • Erhöhter Kommunikationsaufwand: • Erhöhte Anzahl von Objekten

Vorteile der Musternutzung allgemein	Nachteile der Musternutzung allgemein
• Muster bieten eine **Sprache** für das Design • Probleme, Lösungen werden **explizit** gemacht	• Muster sind **keine** fertigen Entwurfslösungen • Muster **ersetzen nicht** Konzept noch sauberen Entwurf

Einstieg ins Testen

Testen – Die - auch mehrfache - Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden

Systematisches Testen – Test, bei dem



(1) die Randbedingungen definiert erfasst sind,
(2) die Eingaben systematisch ausgewählt wurden, und
(3) die Ergebnisse dokumentiert und nach vor dem Test festgelegten Kriterien beurteilt werden

Verifizierung – Prüfen, ob die Ergebnisse einer Phase des Projekts mit denen der vorherigen Phase übereinstimmen

→ "Wurde das System **richtig entwickelt**?"

Validierung – Prüfen, ob das endgültige Ergebnis des Projekts wirklich dem Bedarf des Kunden entspricht

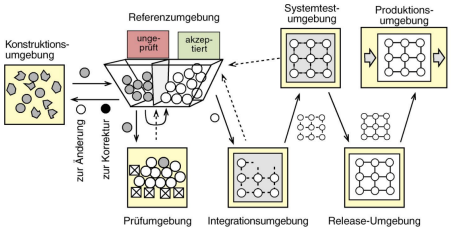
→ "Wurde **das richtige System** entwickelt?"

Qualitätsprioritäten: Richtigkeit, Sicherheit, Leistung, schnelle Entwicklung, geringer Aufwand, Wartungsfähigkeit ...
Testziele: Hoher Anteil getesteter Codezeilen, hoher Anteil getesteter Funktionen, hohe Fehlererkennungsrate, schnelle Tests ...

Arten von Tests

Funktionsprüfung:	**Nicht funktionale Prüfung:**

Unit Testing Integration Testing System Testing - Gesamte Prüfung Acceptance Testing - Anforderungtesting Smoke Testing - Prüfung in Prod-Umgebung	Performance Testing Security Testing Usability Testing Compatibility Testing



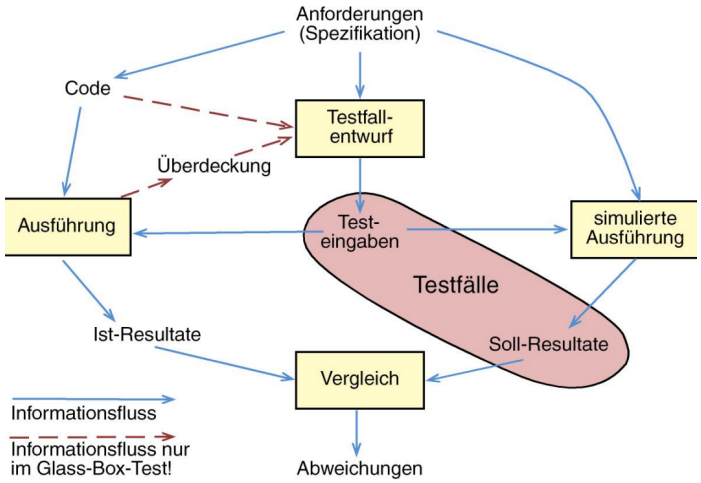
Verwengung von Staging Areas

DEV: Entwicklungsumgebung eines Entwicklers
INT: Integrationsumgebung eines Teams mit Mocks externer Systeme
CON: Konsolidierungsumgebung für Testen mit externen Systemen durch ein Testteam
OPS bzw. PROD: Die Produktivumgebung beim Kunden

Funktionsprüfung

Unit Testing – Testen einzelner Softwaremodule oder Komponenten eines Softwaresystems

→ von den Programmierern des Moduls testgetrieben erstellt



→ Jede Modulfunktion wird durch Testfälle getestet
→ Unit-Tests laufen automatisiert in DEV Staging-Bereich

Drei Gesetze der testgetriebenen Entwicklung
1. keinen Produktionscode, außer um fehlschlagenden Test zu bestehen
2. nur so viel von einem Test, um einen Fehler zu demonstrieren.
3. nur gerade so viel Produktionscode, um den Test zu bestehen.

Integrationstest – Testen einzelnen Softwaremodule im Zusam-menspiel, um spezifische Aufgaben / Aktivitäten durchzuführen

→ Prüfung erfolgt als Kombination aus *automatisierten* und (falls nötig) *manuellen* Funktionstests.
→ Integrationstests laufen im **INT**-Staging-Bereich

Systemtest – Prüfung des System in seiner Gesamtheit auf Fehler

→ Verbund aller Hardware- und Softwarekomponenten des gesamten Systems
→ als **Blackbox-Testmethode** angesehen, in Software auf vom Benutzer Arbeitsbedingungen sowie auf Ausnahmebedingung
→ Systemtests laufen im **CON**-Staging-Bereich (sollte eine vollständige Kopie von OPS bzw. PROD sein.)

Akzeptanztest - Prüfung, ob das System alle Anforderungen des Endkunden erfüllt

→ Akzeptanztests werden im **CON**-Staging-Bereich vom Endkunden ausgeführt (ggf.auch vom Product Owner)

Smoke-Test – Prüfung der Funktionsfähigkeit der OPS/PROD-Umgebung nach Develop einer Software-Version vor Relase

→ Nach den Smoke-Tests startet üblicherweise eine 2-3 wöchige Testphase am LiveSystem

Nicht funktionale Prüfung (oo):

Spezifikationstest (auch: Black-Box-Test):
→ Überprüfen der Eingabe-/Ausgabebeziehungen einer Klasse oder Komponente.
→ Die interne Struktur wird vernachlässigt.
→ Erstellung geeigneter Äquivalenzklassen

> **Äquivalenzklasse** – Bezüglich Ein- und Ausgabedaten ähnliche Objekte, wo erwartet wird, dass sie sich gleich verhalten

---

Beispiel: Anwendungsfall "Veranstaltung finden"
Anwendungsfall: Veranstaltung finden | *Primärer Akteur*: Käufer | *Kontext*: Ticket Informationsschalter | *Bedingungen*: keine
Testenszenario:
1. **Benutzer** wählt Genre und einen Datums- und Uhrzeitrahmen aus. Das System überprüft, dass der Zeitrahmen stimmt
2. Das **System** zeigt das Ergebnis der Abfrage.
3. Der **Benutzer** fordert eine detaillierte Ansicht einer ausgewählten Veranstaltung an.
4. Das **System** zeigt die Detailansicht für die ausgewählte Veranstaltung.

Erweiterungen: 2a. Die Abfrage enthält keine Ergebnisse. Zurück zu 1. oder abbrechen.
Konsequenzen: keine.
Anweisungen: Abfrage durch das Webportal des Systems, Kiosk-Schnittstelle, Kassa-Client (unterstützter sekundärer Akteur)

Nr.	Testfall	Erwartetes Ergebn.	Eingabedaten
1	Entering valid name	Search Result	name="cinema word"
2	Entering valid state	Search Result	state="NÖ"
3	Entering empty state	Exception Message	state="
4	Entering invalid state	Empty Search Result	state="bavaria"
5	Entering valid city	Search Result	city="Hürm"

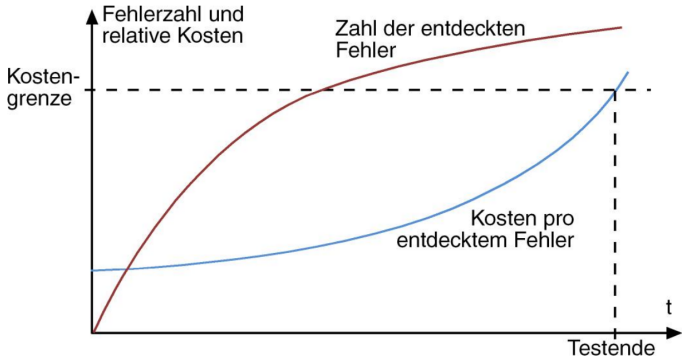
6	**Entering city not in database**	**Empty Search Result**	**city="4444"**

> **Schwellenwert** – Wert, der gerade noch innerhalb oder schon außerhalb eines gültigen Wertebereichs liegt.

→ Falsche Vergleichsoperatoren, z.B. < statt <=, Rundungsfehler,...

Strukturtest (White-Box-Test):
→ Prüfung der *internen Programmstruktur*
→ Struktur des zu untersuchenden Objekts muss bekannt sein
→ Experimentiere: 1. Bestimme alle Ausführungswege und 2. Entdecke mögliche Fehler in einem dieser Wege
→ **Vorteil**: Lage des Fehlers kann sehr genau eingegrenzt werden
→ **Nachteil**: Sehr kosten- und arbeitsintensiv

Grad der Codeabdeckung



C0-Überdeckung oder Knotenüberdeckung: Jede auftretende **Anweisung** im Code muss mindestens **einmal** in einem Testfall durchlaufen werden.

C1-Überdeckung oder Kantenüberdeckung: Jede **Kante** im Kontrollflussgraph muss mindestens **einmal** in einem Testfall durchlaufen werden.

C*infinite*-Überdeckung oder Pfadüberdeckung: Jede (praktisch) mögliche Reihenfolge aller Codeanweisungen müssen von den Testfällen abgedeckt werden.

Bedingungsüberschneidung: Jede Kombinationen der Teilausdrücke logischen Ausdrucks muss in Testfällen überprüft werden.

Kantenüberlappung vs. Zustandsüberlappung IF (A=true) and (B=false) THEN ...
Zwei Testfälle, um eine **Kantenüberlappung** zu erreichen: (A=true) and (B=false) == true | (A=true) and (B=true) == false
Vier Testfälle, um **Bedingungsüberlappung** erreichen: A = true, B = true A = true, B = false A = false, B = true A = false, B = false