

---

# Design Pattern: State

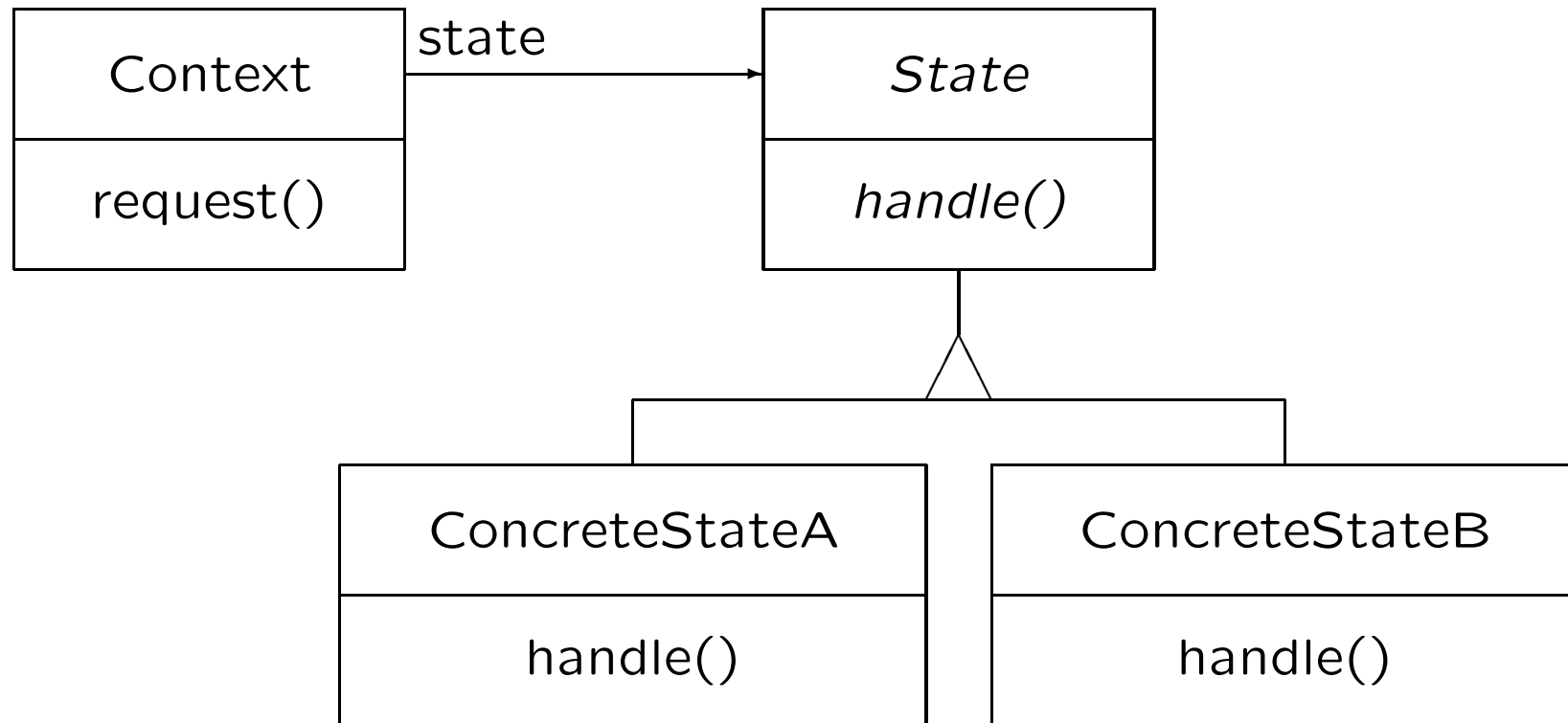
Zweck: Erlaubt Objekt Verhaltensänderung wenn interner Zustand sich ändert; scheint Klasse zu ändern

Anwendungsgebiete:

- Verhalten hängt von Zustand ab, und Verhalten muss sich zur Laufzeit entsprechend dem Zustand ändern.
- Zur Vermeidung bedingter Anweisungen, die vom Objektzustand (oft durch Konstanten dargestellt) abhängen. Jeder Zweig der bedingten Anweisung wird in eigener Klasse dargestellt.

---

# Struktur von State



---

# Auswirkungen von State

- lokalisiert zustandsspezifisches Verhalten und trennt Verhalten in unterschiedlichen Zuständen
  - leicht um Zustände und Zustandsübergänge erweiterbar
  - Code auf viele Klassen verteilt
- macht Zustandsübergänge explizit
  - nur konsistente Zustände durch atomare Übergänge
- gemeinsame Zustandsobjekte möglich

---

# Implementierung von State

- wer definiert Zustandsübergänge?  
Context: Folgezustand nicht zustandsabhängig  
State: starke Abhängigkeiten zwischen Unterklassen
- Sprungtabelle als Alternative:
  - Fokus auf Zustandsübergängen (nicht Verhalten)
  - Sprungtabelle leicht generierbar und änderbar
- Erzeugung der State-Objekte
  - oft reicht eine Instanz pro Klasse (Singleton)
- dynamische Vererbung als Alternative (Self)

---

# Design Pattern: Observer

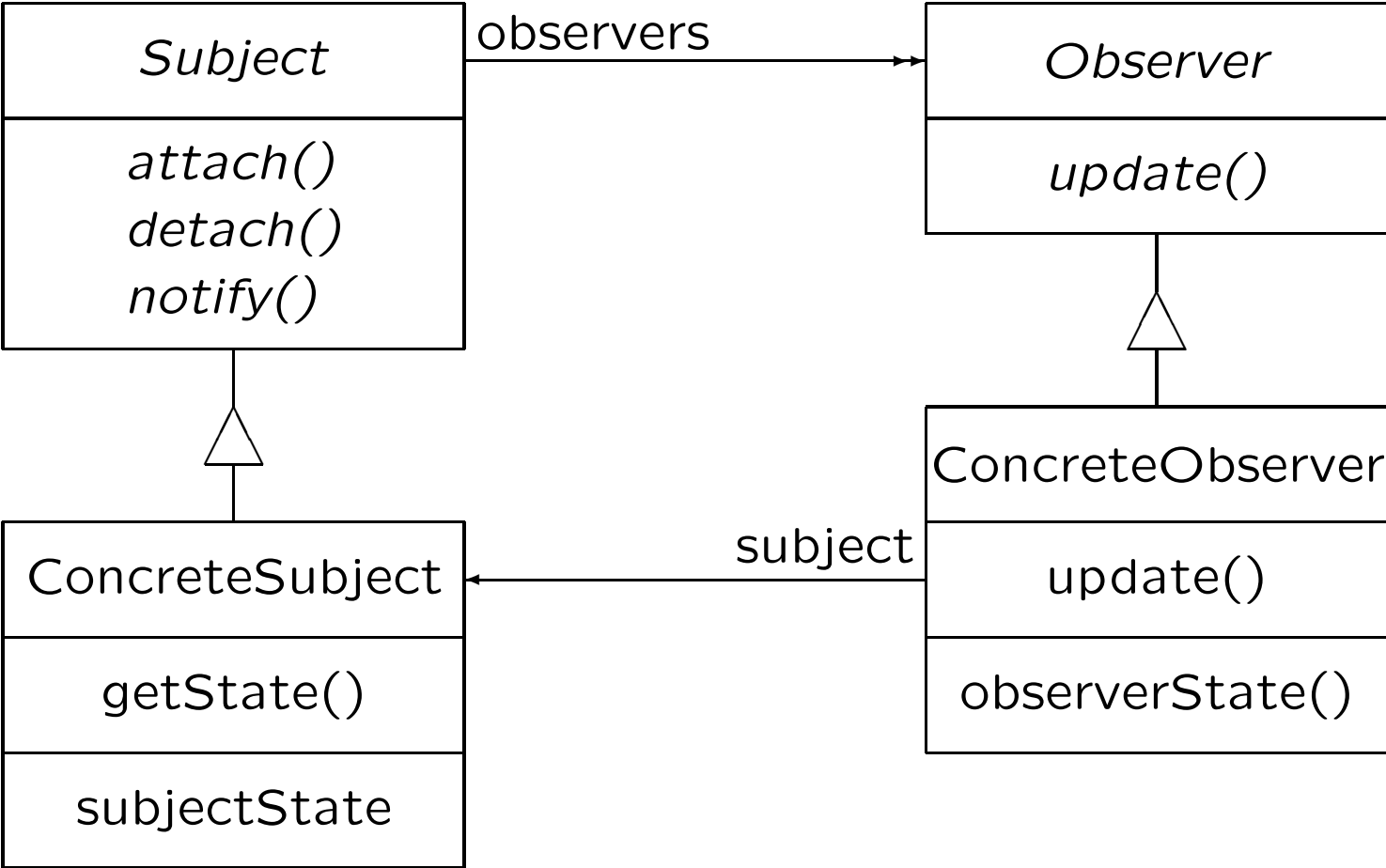
Zweck: Definiert 1-zu-n-Beziehung zwischen Objekten, damit abhängige Objekte benachrichtigt werden, wenn sich der Zustand eines Objekts ändert

Anwendungsgebiete:

- Eine Abstraktion mit zwei Aspekten, einer vom anderen abhängig. Kapselung in getrennte Objekte macht Aspekte unabhängig voneinander änder- und wiederverwendbar.
- Wenn eine Zustandsänderung weitere notwendig macht und statisch unbekannt ist, welche Objekte zu ändern sind.
- Wenn Objekten etwas mitgeteilt werden soll ohne zu wissen, wer diese Objekte sind (keine enge Kopplung).

---

# Struktur von Observer



---

# Auswirkungen von Observer

- abstrakte Kopplung zwischen Subject und Observer
  - können zu unterschiedlichen layers gehören
- broadcast erfolgt automatisch
  - Observer jederzeit hinzufügen und wegnehmen
- unerwartete Updates durch fehlende Information möglich
  - Ursachen unerwünschter Updates schwer zu finden
  - oft hohe Kosten von updates schwer abschätzbar

---

# Implementierung von Observer

- Subject als Argument von update (zur Unterscheidung)
- Wer triggert notify?
  - Client → fehleranfällig;
  - zustandsänd. Subject-Operationen → unnötige updates
- Subject-Zustand soll vor notify konsistent sein
- Subjects entfernen → auf Referenzen in Observers achten
- update mit viel/wenig Information (push/pull-Modell)
- Observers registrieren sich nur für bestimmte Aspekte



---

# Komponentenprogrammierung

- Software aus bestehenden Komponenten zusammensetzen
- gewinnt zunehmend an Bedeutung
- alte Idee, bisher wenig erfolgreich, warum jetzt?
  - mehr Erfahrung, aktives Forschungsthema
  - in Teilbereichen starke Unterstützung durch Industrie
  - Technologie breiter verfügbar  
(Generizität, zahlreiche Komponentenplattformen)
  - Ausbildung

---

# Komponente = Objekt?

- Komponente = Objekt (= verallgemeinertes Modul)
- praktische Unterschiede:
  - Komponenten sollen leicht von einer Umgebung zur anderen portierbar sein
  - „Require Interface“ zusätzlich zu „Deliver Interface“
  - fixe Annahmen über Umgebung (Komponentenmodell)
  - Komponenten komplexer als einfache Objekte
- Komponente  $\neq$  Klasse  
Komponentenbibliothek  $\neq$  Klassenbibliothek

---

# Probleme mit Komponenten

- viele Komponenten „low level“
  - Komponentenprogrammierung bringt wenig
- anwendungsorientierte große Komponenten gewünscht, aber wie oft verwendet? — nur für bestimmte Applikation (oder sogar nur bestimmte Version)
- viele Probleme ungelöst
  - Auffinden, Verteilung, Geschäftsmodelle
  - Beschreibung der Eigenschaften (non-functional prop.)
  - Vertrauen in fremdentwickelte Komponenten
  - Kompatibilitätsprüfung (einfache Typen unzureichend)
  - hot swappable components (Zustand beachten)

---

# Gute Bibliotheken / Komponenten

- systematische Taxonomie
  - sonst Erlernen zu teuer
  - Verhalten besser einschätzbar (Zusicherungen, Namen)
- so generisch wie möglich
  - anwendbar in vielen Bereichen
- so effizient wie möglich
  - sonst nicht verwendet
- Verständlichkeit absolut notwendig
  - sonst nicht verwendet

---

# Persistente Objekte (einfache Form)

Objekte persistent, wenn in einer Datei/Datenbank gespeichert und länger als Programm existent

**Einfachste Form:** Objekt zu (char \*) konvertiert, als String gespeichert, beim Einlesen zurückkonvertiert

- einfach, aber funktioniert nicht mit Zeigern (vorher auflösen) und ist nicht portabel (Programmänderungen)

Standard-Persistenzmechanismen funktionieren ähnlich, aber

- Abhängigkeit von Hardware, Compiler, Bibliotheksversion vermieden — Programmänderungen bleiben problematisch
- Fehleranfälligkeit reduziert

---

# Persistente Objekte

meist sinnvoll, externe von interner Darstellung zu trennen:

- internes Format leichter änderbar
- leichter portierbar
- Zeiger (Referenzen) explizit darstellbar

Konvertierung meist kontextabhängig

gemeinsame Oberklasse aller Objekte, die persistent gemacht werden können, nur begrenzt sinnvoll

Datenbankanbindung bei größeren Datenmengen

---

# Konzept versus Datenstruktur

Beispiel: Implementierung einer Liste

Variante A: Liste als Konzept; Listenelemente privat

Variante B: Klasse beschreibt Listenelement (Datenstruktur)

- B einfacher zu implementieren
  - A empfohlen, da schwächere Objektkopplung (B führt leicht zu zusätzlichen Parametern)
  - Probleme bei der Verwendung von B werden normalerweise früh erkannt, Refaktorisierung führt zu A
- falsche Denkansätze rasch erkannt und beseitigt wenn Architekt = Programmierer, sonst problematisch

---

# Globale und statische Variablen

- globale Variablen böse, statische Variablen fast so böse
- statische Variablen in „templates“ fehleranfällig  
(homogene versus viele Arten heterogener Generizität)
- Konstante (statische Variablen) bilden oft Ausnahme:  
Vermeidung starker Kopplung schränkt Änderbarkeit ein
- statische Variablen sinnvoll, wenn tatsächlich als Variablen  
der Klassen selbst gesehen
- Intention unklar wenn Architekt  $\neq$  Programmierer



---

# Zugriff auf Variablen

- public Variablen machen Codeänderungen schwierig
  - public Variablen vermeiden, sowieso klar
- Invarianten auf von außen geschr. Variablen unhandlich
  - Variablen nur über „this“ schreiben, nicht jedem klar
- Vermeidung von public Variablen durch „get“ und „set“
  - Änderungen leichter da Methoden überschreibbar
  - klare Vor- und Nachbedingungen
  - starke Objektkopplung bleibt
  - besseres Design überlegen

---

# Binäre Methoden – Lösungsvergleiche

- dynamischer Typvergleich  
→ akzeptabel, da nur eigener Typ vorkommt
- Kombination Überladen und dynamischer Typvergleich  
→ effizient wenn Typ statisch bekannt
- Multimethoden bzw. Visitor Pattern → meist overkill
- einfache Generizität  
→ selten eine Lösung, da Typ statisch unbekannt
- gebundene Generizität  
→ Lösung, wenn von Sprache und Klassen unterstützt  
(aber Probleme bei tiefen Hierarchien)