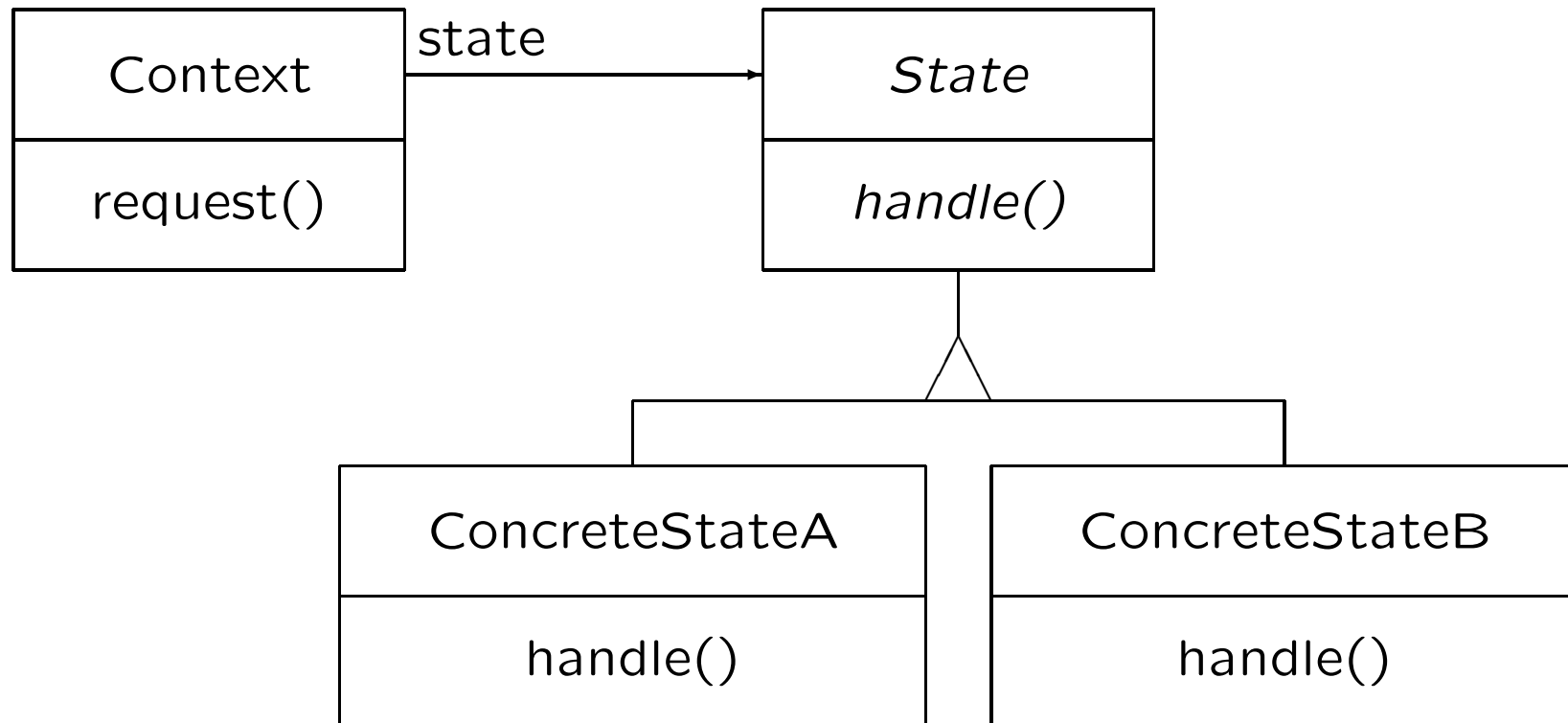

Design Pattern: State

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)

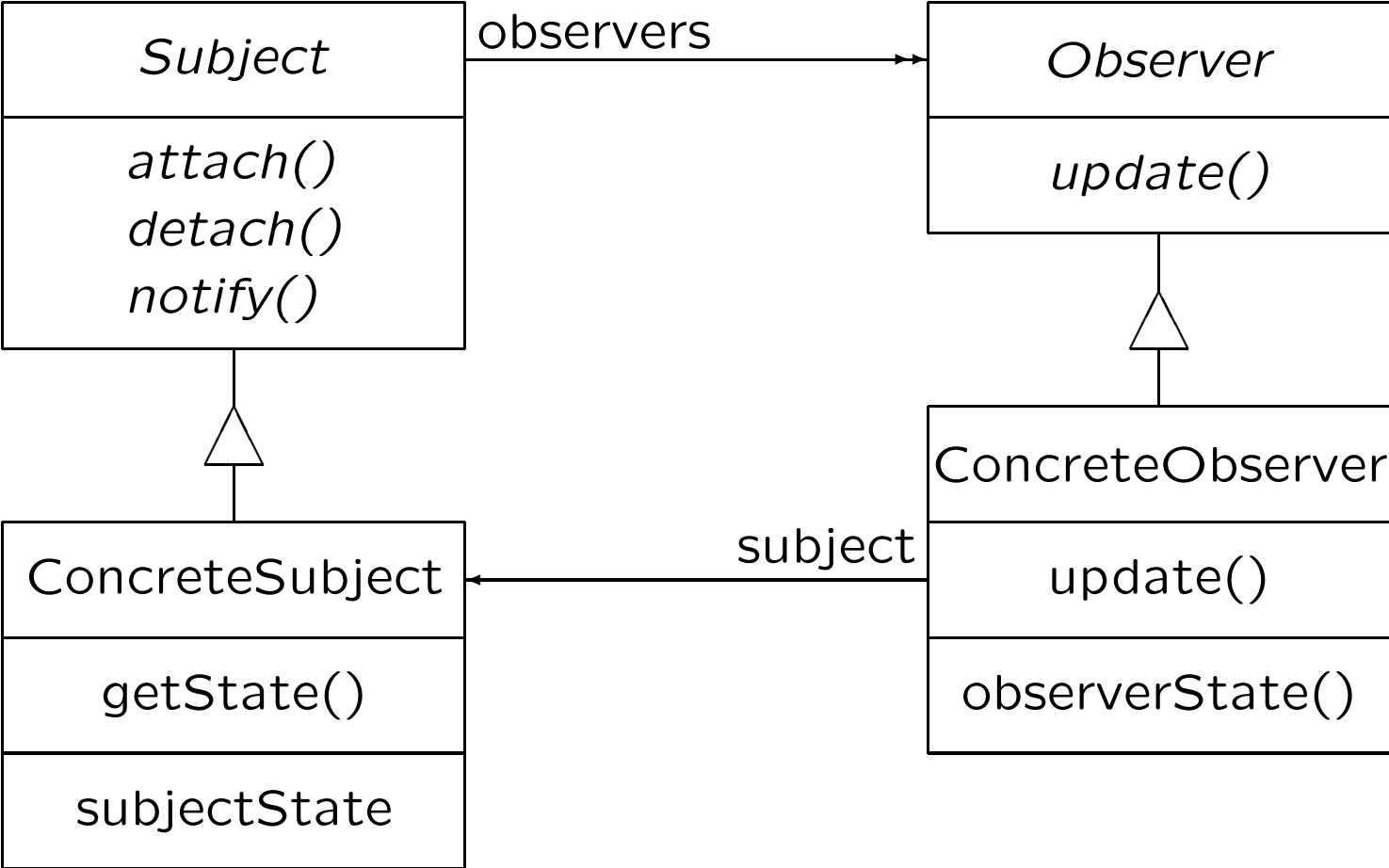
Design Pattern: Observer

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

- Referenzen zu Observers in Subject / globale hash table
- Subject als Argument von update (zur Unterscheidung)
- Wer triggert notify? Client → fehleranfällig;
zustandsändernde Operationen → viele 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?

- grundlegende Sichtweise: Komponente = Modul = Objekt
- praktische Unterschiede:
 - Komponenten sollen leicht von einer Umgebung zur anderen portierbar sein
 - „Require Interface“ zusätzlich zu „Deliver Interface“
 - fixe Annahmen über Umgebung (Komponentenmodell)
 - Komponenten viel 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 noch ungelöst
 - Auffinden, Verteilung, Geschäftsmodelle
 - Beschreibung der Eigenschaften
 - Vertrauen in fremdentwickelte Komponenten
 - Kompatibilitätsprüfung (einfache Typen unzureichend)
 - hot swappable components (Zustand beachten)

Forderungen an gute Bibliotheken

- systematische Taxonomie für Bibliothek
 - sonst Erlernen der Bibliothek zu teuer
 - Verhalten besser einschätzbar (Zusicherungen, Namen)
- so generisch wie möglich
 - Anwendbar in vielen Bereichen
- so effizient wie möglich
 - sonst nicht verwendet
- Bibliotheken müssen verständlich sein
 - 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
- nicht portabel, da Speicherformat von Hardware, Compiler, Programmversion (Bibliotheksversion) abhängig
- fehleranfällig weil oft nicht im Detail verstanden

Persistenzmechanismen von Standardbibliotheken kaum besser

Persistente Objekte

meist sinnvoll, externe von interner Darstellung zu trennen:

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

Funktionen zur Konvertierung zwischen interner und externer Darstellung meist kontextabhängig

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

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, und Refaktorisierung führt zu A

→ Denken in Konzepten

Globale und statische Variablen

- globale Variablen bewirken, dass Änderungen beliebige andere Programmteile beeinflussen können
- statische Variablen von vielen Objekten manipulierbar
- statische Variablen in „templates“ fehleranfällig (homogene versus viele Arten heterogener Generizität)
- Konstante bilden oft Ausnahme (Vermeidung starker Objektkopplung würde Änderbarkeit einschränken)
- statische Variablen sinnvoll, wenn tatsächlich als Variablen der Klassen selbst gesehen

Zugriff auf Variablen

- public Variablen machen Implementierungsdetails sichtbar
 - nachträgliche Codeänderungen schwierig
 - public Variablen vermeiden
- Variablen möglichst nur über „this“ schreiben
- Vermeidung von public Variablen ist durch „get“- und „set“-Methoden immer möglich
 - Änderungen leichter als bei Variablen, da Methoden in Unterklassen überschreibbar
 - starke Objektkopplung bleibt
 - besseres Design überlegen