Impl. dyn. Binden bei Einfachvererbung

- jedes Objekt enthält Zeiger auf dynamischen Typ
- Typ enthält Array von Zeigern auf Methoden (VFT)
- Methode intern als Array-Index dargestellt,
  Variable als Offset von Objektadresse
- dyn. Binden: Methode in Array-Eintrag des dyn. Typs
  Variablenzugriff über Offset (dyn. Typ nicht nötig)
- Einfachvererbung: Indexe und Offsets bleiben erhalten
  zusätzliche Einträge und Offsets hinten angehängt
  ererbte Methode = kopierter Array-Eintrag
  überschriebene Methode = überschriebener Array-Eintrag

Impl. dyn. Binden mit Interfaces

- einfache Impl. für Mehrfachvererbung ungeeignet
  (unterschiedliche geerbte Array-Indexe für selbe Methode)
- Array-Eintrag für jedes implementierte Interface
  (zeigt auf Interface, sonst wie Methodenzeiger)
- Interface enthält Array von Methodenzeigern
  (und Variablenoffsets wenn unterstützt)
- eine Indirektion mehr, Variablenzugriffe teurer
- Einfachvererbung von Interfaces durch hinten anhängen,
  Mehrfachvererbung durch zusätzliche Interfaces in Array
- manchmal Interface-Array direkt in Objekt

Impl. dyn. Binden, Mehrfachvererb. 1

- im Prinzip Techniken für Interfaces verwendbar
  wenn nötig mehrere Arrays (VFT = virtual function table)
- Objektlayout ererbter Klassen in C++ direkt übernommen
  (für up-cast auf Objekten notwendig)
- up-cast ändert Zeiger um Konstante „delta“ auf Anfang des Teilobjekts
- Anfang jedes Teilobjekts zeigt auf eigenen VFT
- bei Vergleichen auf Identität „delta“ berücksichtigen

Impl. dyn. Binden, Mehrfachvererb. 2

- Methodenaufruf bei Mehrfachvererbung (allgemein):

```
1.   load [objectReg + #VFToffset], tableReg
2.   load [tableReg + #deltaOffset], deltaReg
3.   load [tableReg + #selectorOffest], methodReg
4.   add objectReg, deltaReg, objectReg
5.   call methodReg
```

- „delta“ meist 0 → erlaubt Optimierung mit „thunks“:
  - — Befehle 2 und 4 weglassen
  - — Sprung auf „thunk“ (= überschriebene Methode):

```
thunk: add objectReg, #delta, objectReg
       jump #method
```

Optimierungsmöglichkeiten

- wenn Methode statisch bekannt:
  - direkter Sprung statt über VFT
  - „inlining“ und weitere Optimierungen möglich
- Methode eher statisch bekannt wenn Compiler komplette Typhierarchie kennt
- teilweises „inlining“ (Spezialisierung) durch „thunks“
- durch Wachsen in beide Richtungen und bei statisch bekannter Typhierarchie „thunks“ fast immer vermeidbar
- dynamisches Caching: zuerst auf selbe Methode wie bei letztem Aufruf springen, dann schauen ob Typ passt und nötigenfalls weiterspringen

Implementierung Exception Handling

- Compiler erzeugt Tabelle, die Adressbereichen (Blöcken) Exception-Handler zuordnet (für jeden Exception-Typ)
- nach Auftritt wird Handler für aktuellen Instruction Pointer (IP) gesucht und (wenn vorhanden) darauf gesprungen
- existiert kein passender Handler, wird aus aktueller Routine zurückgekehrt und mit neuem IP (Aufrufstelle) gesucht
- für Code nach Rückkehr aus „main“ existiert Handler
- kein Overhead ohne Exception, Handler-Suche billig, Objekterzeugung manchmal teuer

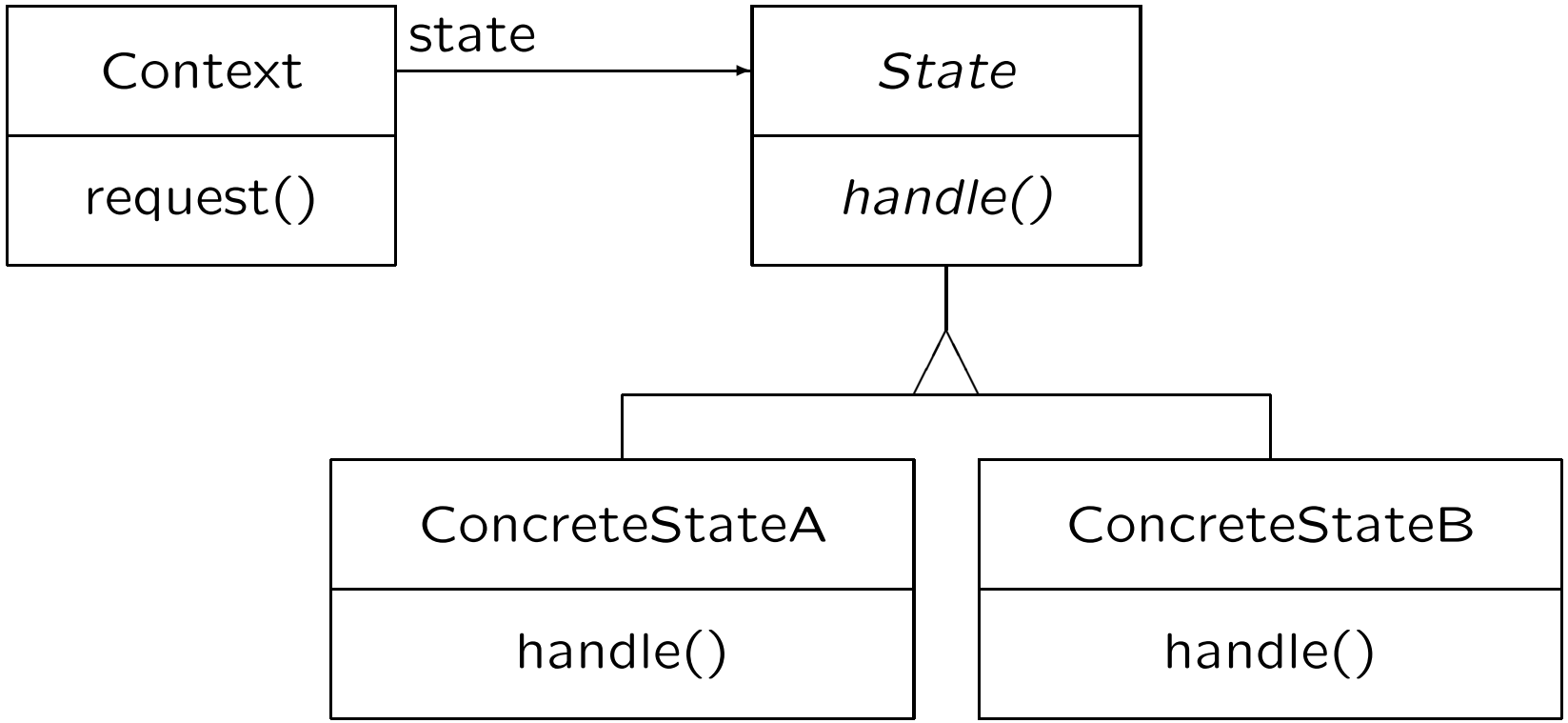
Design Pattern: State

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

  $\rightarrow$ leicht um Zustände und Zustandsübergänge erweiterbar

  $\rightarrow$ Code auf viele Klassen verteilt

- macht Zustandsübergänge explizit

  $\rightarrow$ 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)

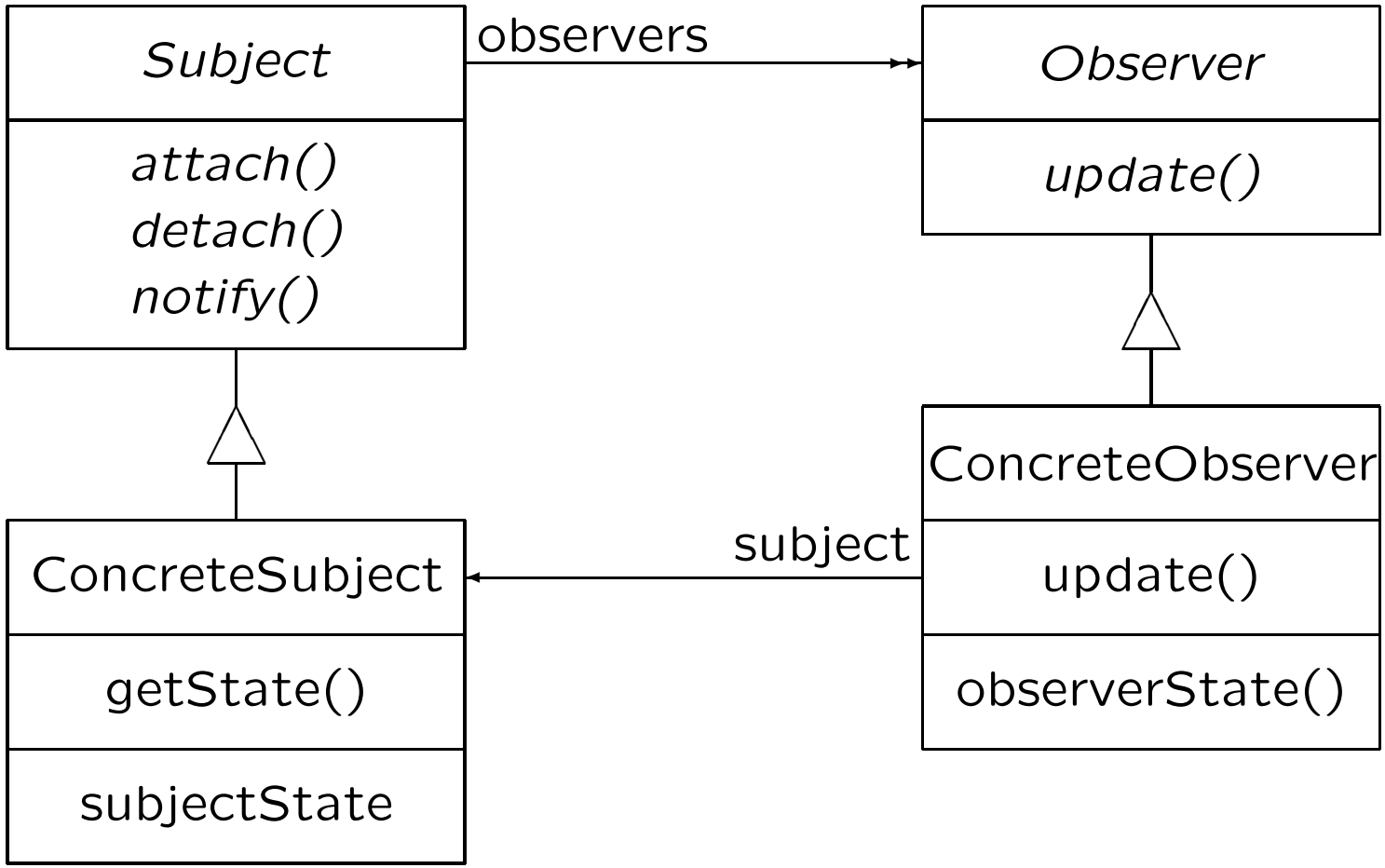
Design Pattern: Observer

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

Implementierung von Observer

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