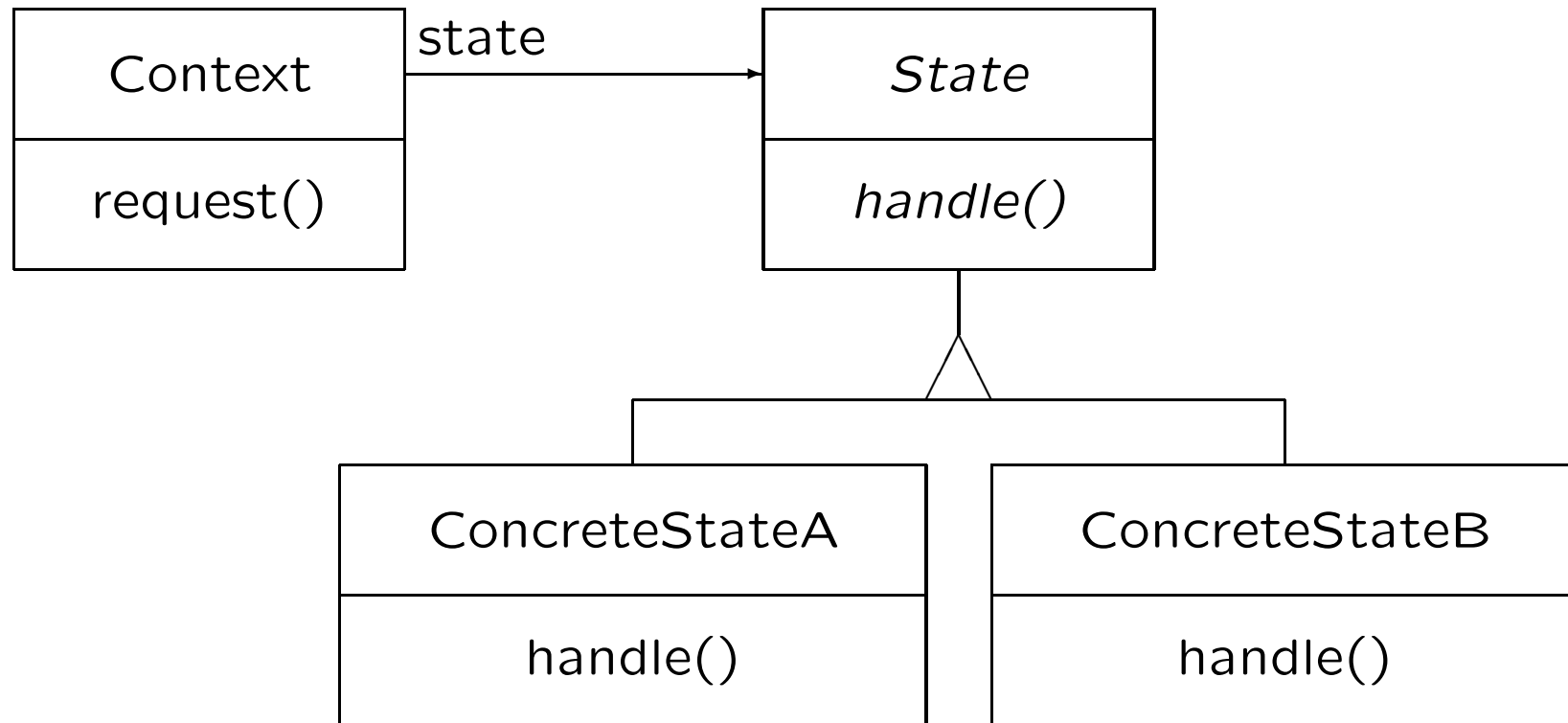

Design Pattern: State

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)

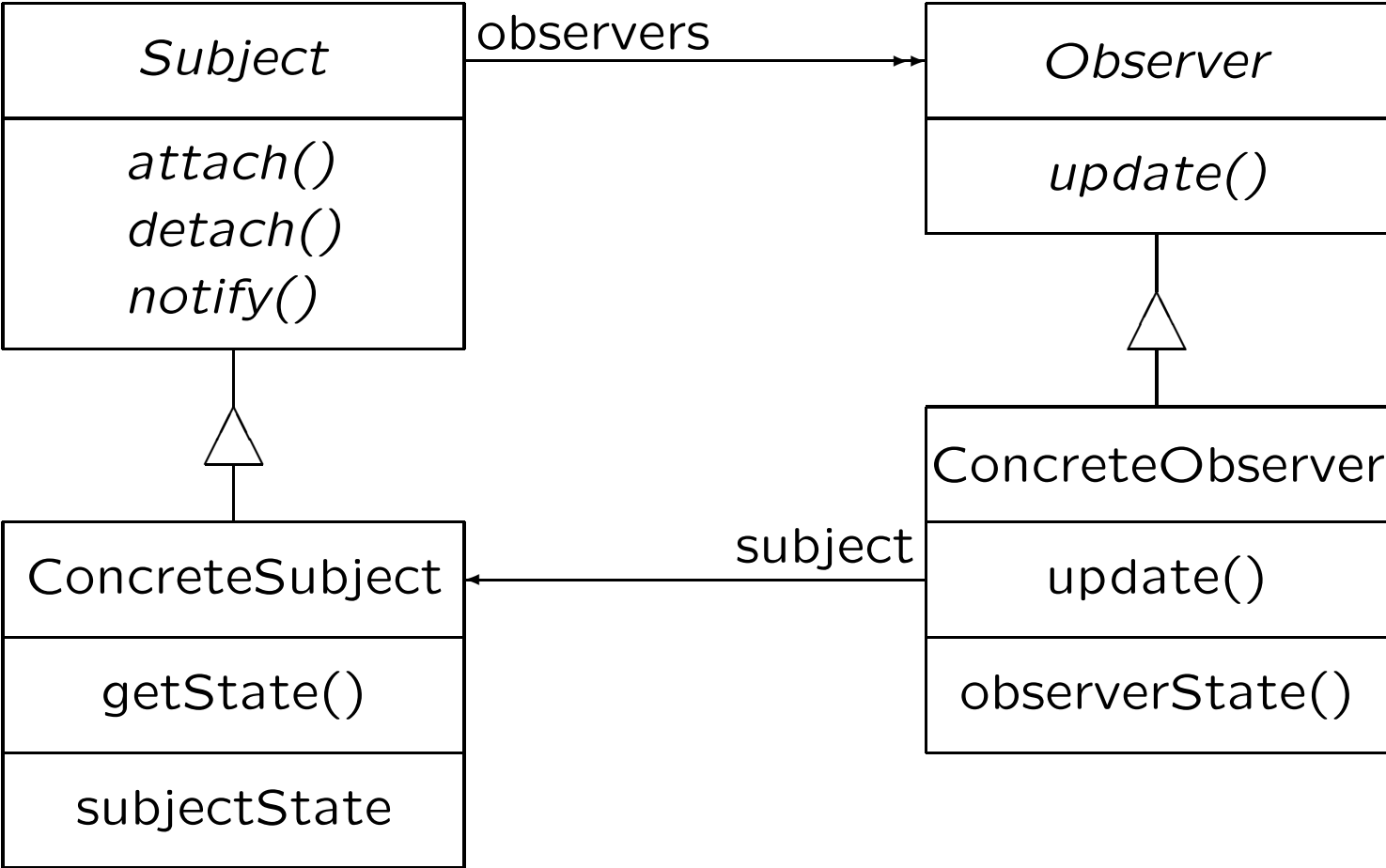
Design Pattern: Observer

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 Zustandsänderung andere notwendig macht und 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

Ebenen der Interprozesskommunikation

- einfacher Input/Output – unsynchronisiert
- „mutual exclusion“
- serialisierbare Kommunikation
- atomare Aktionen
- Transaktionen

wenn mehr als „mutual exclusion“ gefordert, muss sich Programmierer selbst darum kümmern

Aspekte der Kommunikation

Für die Ebenen der Interprozesskommunikation und der Sicherheit vor unerwünschten Zugriffen gelten:

- hohe Ebene: teuer und ineffizient
- niedrige Ebene: fehleranfällig, oft inakzeptabel
- Optimum hängt von der Anwendung ab
- Programmierer/Anwender sollen wählen können

Reflektive Programmierung

Reflektion: die Sprache wird dynamisch verändert

- lässt dem Programmierer viele Freiheiten
- fehleranfällig
- für aspektorientierte Programmierung geeignet

Änderungen der Semantik eines Programms möglich durch

- Änderungen des Programms
- Änderungen der Sprachen bzw. Standardbibliotheken
- Parameterisierte Sprachen bzw. Bibliotheken

OO Arten der Reflektion

- interne Klassenstruktur sichtbar machen (Metadaten)
- Klassen zur Laufzeit hinzufügen, Typinfo. verwenden
- Klassen dynamisch erzeugen bzw. verändern
- „message passing mechanism“ oder andere Sprachkonzepte der untersten Stufe sichtbar machen und verändern

Alternativen zur Reflektion

Precompiler vor Programmübersetzung ausführen:

- effizient zur Laufzeit
- einfache Änderungen mit kleinem Aufwand
- größere Analysen können teuer sein
- Laufzeitinformation nicht verfügbar

Programmiersprache ändern oder erweitern

- nur für große Änderungen sinnvoll
- interne Informationen im Compiler wiederverwendbar
- nicht portabel (mit Ausnahmen)
- einfach, wenn Erweiterung nur mit Bibliotheken

Attribute in C#

- Attribut = Objekt, das mit Programmelementen verknüpfte Daten darstellt

- verknüpfbar mit Klassen, Methoden, Parametern, etc.:

```
[Serializable]  
class MySerializableClass { ... }
```

- intrinsic (in .NET integriert) versus benutzerdefiniert

- Beispiel für Verwendung eines benutzerdef. Attributs:

```
[BugFixAttribute("Stefan", "17.3.2006", Comment="OK")]  
class MyTestClass { ... }
```

- über Reflection zur Laufzeit zugänglich

Beispiel für Attributdefinition

```
[AttributeUsage(AttributeTargets.Class,AllowMultiple=true)]
public class BugFixAttribute : System.Attribute {
    public BugFixAttribute (string programmer, string date) {
        this.programmer = programmer;  this.date = date);
    }
    public string Comment {
        get { return comment; }
        set { comment = value; }
    }
    public string Programmer { get { return programmer; } }
    public string Date { get { return date; } }
    private string programmer, date, comment;
}
```

Annotationen in Java

- entsprechen ungefähr Attributen von C# – Beispiel:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String prog(); String date(); String comment();
}
...
@BugFix(prog="Stefan", date="17.3.2006", comment="OK")
class MyTestClass { ... }
```

- SOURCE Annotationen ersetzen javadoc Annotationen
- eigentlich mächtige als Attribute, aber zur Laufzeit derzeit kaum verwendet (nicht in Standardbibliotheken)

Dynamische Erweiterung von Klassen

- dynamische Sprachen unterstützen Klassenerweiterung zur Laufzeit, statische Sprachen nicht direkt – Auswirkungen?
- Decorator und Proxy erlauben dynamische Erweiterung
- durch „final“ nur ohne Ersetzbarkeit verhinderbar
→ Erweiterbarkeit auch in statischen Sprachen Normalfall
- direkte Klassenänderung in dyn. Sprachen etwas effizienter als Decorator/Proxy (Laufzeit + Programmieraufwand)
- statisches Konzept effizienter wenn keine Klassenänderung nötig (auch in dynamischen Sprachen)