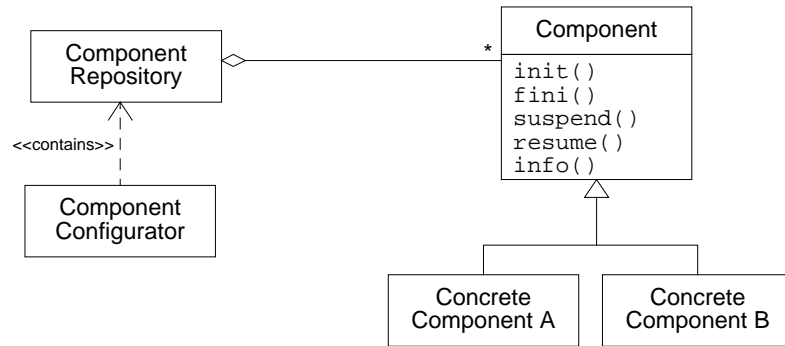


1 Service Access and Configuration Patterns (2)

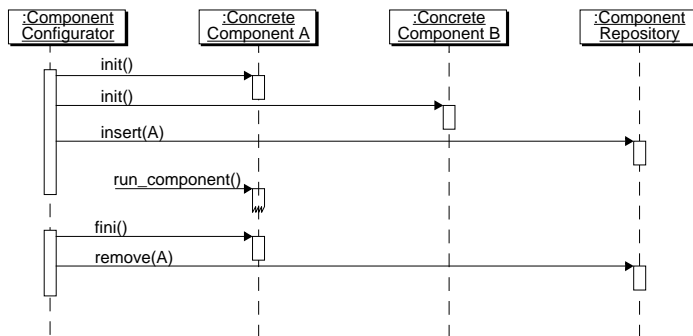
■ Component Configurator (Service Configurator)

- dynamisches Binden und Entfernen von Komponenten einer Anwendung
- Realisierung über DLL und Austausch von Komponenten zur Laufzeit



- - > = Abhängigkeit (Dependency)

- Allgemeine Komponentenschnittstelle und konkrete Komponente werden getrennt.
 - über die allgem. Komponentenschnittstelle können konkrete Komponenten geladen, initialisiert und entfernt werden - unabhängig von der jeweiligen Funktionalität (Abstraktion von der DLL-Schnittstelle).
 - eine konkrete Komponente enthält Standardschnittstellen zur Konfiguration (Oberklasse) und die anwendungsspezifische Schnittstelle (Unterklassen)
- Ein *Component Repository* verwaltet alle konkreten Komponenten
- Der *Component Configurator* verwaltet konkrete Komponenten mit Hilfe des Repositories.



1 ... Service Access and Configuration Patterns

■ Interceptor

- Erweiterung von Ausführungsumgebungen: dynamisches Hinzufügen von Diensten und automatische Benachrichtigung oder Zwischenschaltung bei bestimmten Ereignissen

➤ Beispiel: Security Service, Logging Service

- Problem: Ausführungsumgebungen (Object Request Broker, Application Server, etc. - siehe spätere Kapitel über CORBA und EJB) können nicht für alle evtl. benötigten Dienste vorbereitet sein.
 - es sollen z. B. bestimmte Zugriffsschutzmechanismen bei Methodenaufrufen ohne Änderung der Architektur der Ausführungsumgebung hinzugefügt werden können
 - oder Objekte sollen auf mehrere Rechner repliziert werden und die Methodenaufrufe sollen entsprechend verteilt und die Ergebnisse zusammengeführt werden - ohne die Anwendung oder die Ausführungsumgebung direkt ändern zu müssen
- Lösung: Anwendungen können spezielle Dienste (=Objekte) bei der Ausführungsumgebung registrieren. Bei bestimmten "Ereignissen" (z. B. abgehender oder ankommender Methodenaufruf, Auftreten einer Exception) wird zunächst an dem registrierten Objekt eine Methode aufgerufen.

■ Extension Interface

➤ mehrere Schnittstellen für eine Komponente

➤ Beispiel: Standard-Schnittstelle, Debug-Schnittstelle, Administrations-Schnittstelle, ...

- Problem: Software-Komponente wird im Laufe der Zeit weiterentwickelt - existierende und genutzte Schnittstellen soll aber stabil bleiben.
- Lösung: Nicht eine Klasse mit einer großen Schnittstelle, sondern viele kleine Schnittstellen, die semantisch zusammenhängende Methoden zusammenfassen. (vgl. Abschnitt C.6.6 - Typhierarchie und 1:1-Verhältnis Klasse-Typ)
 - Component Factory zur Instantiierung einer Komponente liefert allgemeine Schnittstelle über die weitere Schnittstellen (Extension Interfaces) angefordert werden können.
 - Jede Funktionsgruppe der Komponente hat eigene Schnittstelle, die über die allgemeine Schnittstelle abgerufen werden kann.
- Diese Vorgehensweise ist z.B. auch bei Microsoft-COM-Objekten so realisiert.

2 Event Handling Patterns

■ Reactor (Dispatcher, Notifier)

- Reactor nimmt Ereignisse von mehreren Quellen an und verteilt sie kontext-abhängig an registrierte Event-Handler
- Problem:
Ereignis-getriebene Anwendung (typischer Server) erhält gleichzeitig mehrere Anfragen, arbeitet sie aber synchron und hintereinander ab. Naheliegende Lösung mit mehreren Threads wird wegen des Koordinierungsbedarfs sehr leicht zu komplex.
- Lösung:
 - es wird an einer oder mehrerer Ereignis-Quellen (z. B. Sockets) synchron auf Anfragen gewartet.
 - für jeden Dienst des Servers gibt es einen eigenen *Event Handler* der sich an einem zentralen *Reactor* registriert.
 - *Reactor* nutzt einen *Synchronous Event Demultiplexer (SDM)* (vgl. UNIX-select()), um auf Ereignisse von mehreren Quellen gleichzeitig zu warten. Tritt ein Ereignis ein, informiert der *SDM* den *Reactor*, der daraufhin den für diesen Ereignis-Typ registrierten Event-Handler zur Bearbeitung aufruft. Nach der Bearbeitung wartet der reactor wieder am *SDM*.

■ Proactor

- asynchrone Behandlung lang-dauernder Aktionen gekoppelt mit anwendungs-synchronisierter Ergebnisbehandlung
- nutzt Vorteile von Nebenläufigkeit und vermeidet die Nachteile
- Problem:
Eine verteilte Anwendung startet mehrere asynchrone (nebenläufige) Bearbeitungen und muss nach deren Ende die Ergebnisse bearbeiten.
- Lösung:
Aufteilung in einem asynchronen Teil und einen *completion handler*. Am Ende der asynchronen Bearbeitung aufgerufen wird ein *completion event* in eine *completion event queue* eingetragen. Der *Proactor* sorgt dann für die Ausführung der zugehörigen *completion handler*.

■ Asynchronous Completion Token

- automatische Ergebnisbehandlung asynchroner Dienstabfragen

■ Acceptor-Connector

- Entkoppeln des Verbindungsaufbaus zwischen gleichberechtigten Diensten von der nachfolgenden Interaktion

3 Synchronization Patterns

■ Scoped Locking (Synchronized Block, Guard, Execute Around Object)

- atomatisches Belegen und Freigeben eines Locks beim Betreten bzw. Verlassen eines Abschnitts – unabhängig vom Pfad beim Verlassen

■ Strategized Locking

- Parametrierbarer Synchronisationsmechanismus

■ Thread-Safe Interface

- Locking durch Schnittstellen-Methoden an der "Grenze" einer Komponente
- Komponenten-interne Aufrufe arbeiten ohne Locking

■ Double-Checked Locking Optimization

- Thread "merkt sich" welche Locks er schon hat

4 Concurrency Patterns

- Active Object
 - entkoppelt Methodenaufruf durch Client-Thread von Methodenausführung in Server-Thread(s)
- Monitor Object
 - vermeidet Nebenläufigkeit innerhalb eines Objekts
- Half-Sync/Half-Async
 - Vermittlung zwischen synchronen Dienstaufrufen "von oben" und asynchrone Ereignissen "von unten"
- Leader/Followers
 - Menge von Threads wartet auf Ereignisse. Leader-Thread nimmt Ereignis auf und bearbeitet es, der erste der Follower wird zum neuen Leader
- Thread-Specific Storage
 - globaler Name für Thread-lokale Daten (z. B. errno)