F Design Patterns für nebenläufige und verteilte Objekte

F.1 Überblick

- Motivation
- Standardprobleme
- Design Patterns für nebenläufige und verteilte Objekte

F.2 Literatur

SSRB04. Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: Pattern-Oriented Software Architecture. Volume 2 - Patterns for Concurrent and Networked Objects. Wiley, Chichester, 2004.

. Douglas Schmidt: http://www.cs.wustl.edu/~schmidt/

F.3 Motivation

★ Typische Probleme bei nebenläufiger und verteilter Software

- Inhärente und unbeabsichtigte Komplexität
  - ➤ bedingt durch Verteilung
    - – Inhärente Komplexität durch Umgang mit Teilausfällen des Systems, Verteilten Verklemmungen und End-to-End Quality-of-Service Anforderungen. Je komplexer das verteilte System ist, desto umfangreicher und schwieriger werden die mit diesem Punkt verbundenen Probleme und ihre Lösungen.
  - ➤ bedingt durch Einschränkungen in Werkzeugen und schlechte Unterstützung durch Sprachen
    - – Unbeabsichtigte Komplexität z. B. durch Probleme mit nicht-portablen Anwendungsschnittstellen oder schlechte Debug-Unterstützung im verteilten System. Häufig entsteht Komplexität auch, wenn Entwickler einfache Sprachen und Werkzeuge für eine Systementwicklung auswählen, die dann für beim Einsatz für komplexe nebenläufige und verteilte Software nicht angemessen sind.
- Nicht-adäquate Methoden und Techniken
  - ➤ berücksichtigen Besonderheiten von Verteilung nicht
    - – Verbreitete Software-Analyse und -Design Methoden fokussieren primär auf sequentiell strukturierte Anwendungen. Aspoekte von Nebenläufigkeit und Verteilung werden der Intuition des Software-Designers überlassen.
- Häufige Neu-Erfindung grundlegender Konzepte und Techniken

F.4 Standardprobleme

- Service Access and Configuration
  - ➤ Kommunikation (low level oder durch Middleware unterstützt)
    - Explizite Kommunikation basierend auf low-level-Schnittstellen führt zu unbeabsichtigter Komplexität in vielfacher Hinsicht: Entwickler muss viele low-level details bei der Programmierung berücksichtigen, es werden immer wieder higher-level Abstraktionen neu erfunden und realisiert, Programmierung ist fehleranfällig, wenig portabel, hoher Einarbeitungaufwand für Entwickler aufgrund vieler Details, Skaliert nicht mit zunehmender Komplexität des Gesamtsystems
    - Implizite Interaktion ist für den Entwickler einfacher zu handhaben. Ohne die Möglichkeit einer flexiblen Anpassung der unterstützenden Middleware können spezielle Anforderungen einer Anwendung (z.B. in Bezug auf Quality-of-Service) nicht erfüllt werden.
  - ➤ Konfiguration (dynamische Änderungen eines Dienstes)
    - Idealerweise sollten Dienste, die verteilte Anwendungen unterstützen anwendungsspezifisch konfiguriert und ggf. auch zur Laufzeit ausgetauscht werden können. Solche Konfigurationen und Konfigurationsänderungen sollten unabhängig von der Anwendungsfunktionalität sein.
    - Grundlegender Mechanismus hierfür ist dynamisches Binden. Die Schnittstellen zum Umgang mit DLLs ist aber häufig zu niedrig angesetzt. Es fehlen Strategien, wie man mit diesen Mechanismen konzeptionell in einer Anwendung umgeht.
- Event Handling
  - ➤ Reaktion auf externe und interne Ereignisse
    - Lange Zeit beschränkte sich der Umgang mit externen Ereignissen auf low-level Betriebssystemprogrammierung (Behandlung von Interrupts von Geräten). Normale Anwendungsprogramme waren sequentiell strukturiert und warteten ggf. auf an einer Systemschnittstelle auf ein Ereignis (normaler Funktionsaufruf der erst nach Eintreffen und Bearbeitung des Signals im Betriebssystem zurückkehrt).
    - In nebenläufigen und verteilten Anwendungen ist die Signalisierung von bestimmten Ereignissen an einen parallelen Ablauf auch auf Anwendungsebene eine häufig auftretende Situation. Low-level Mechanismen wie Signalbehandlung oder UNIX-select() führen führen schnell zu komlexen und unübersichtlichen Lösungen.
- Concurrency and Synchronization
  - ➤ Strukturierung und Synchronisation von Threads
    - Low-level Thread-Programmierung ist komplex und führt zu unübersichtlicher Software
    - Analog ist der Umgang mit einfachen Koordinierungsmechanismen wie lock/unlock oder Bedingungsvariablen zwar flexibel, führt aber leicht zu Feehlern und ebenfalls zu unübersichtlichen Softwarelösungen

---

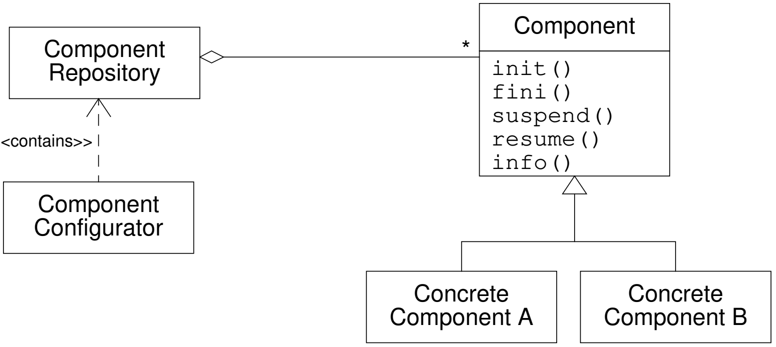
F.5 Design Patterns

1 Service Access and Configuration Patterns

- Wrapper Facade
  - ➤ kapselt Funktionen und Daten von nicht-objektorientierten Schnittstellen in Klassen
  - ➤ Beispiel: Management von TCP-Verbindungen durch *handle*-Objekte
    - Anwendungskontext:
      Wart- und erweiterbare Anwendungen, die Mechanismen und Dienste von bereits existieren nicht-objektorientierten APIs verwenden müssen.
    - Problem:
      OO Anwendungen nutzen oft nicht-objektorientierte Bibliotheksfunktionen oder Systemschnittstellen (z. B. Socket- oder Thread-Schnittstellen, Datenbankanbindungen).
      - – nicht-objektorientierte Kode-Sequenzen sind weniger kompakt und aussagekräftig als objektorientierter Kode mit Konstruktoren, Exceptions oder Garbage Collection.
      - – der Kode ist fehleranfälliger (viele Details zu beachten, z. B. Network-Byteorder, Strukturinitialisierungen, korrekte Fehlerbehandlung bei jeder Funktion)
      - – Portabilität - z. B. zwischen Solaris, Linux und Windows - ist im Detail problematisch. POSIX-Schnittstelle wird prinzipiell überall unterstützt wird, aber im Detail existieren kleine Unterschiede, die an der Schnittstelle nicht unmittelbar zu erkennen sind.
      - – Systemabhängiger Code wird durch #ifdefs geklammert - die Software wird dadurch oft völlig unleserlich und extrem schwer wartbar.
    - Lösung:
      - – Nicht-objektorientierte Schnittstellen werden nicht direkt angesprochen.
      - – Für jede Menge zusammengehörender Funktionen wird eine *wrapper facade* Klasse bereitgestellt:
        - INET_Addr für die Funktionen rund um IP-Addr-Strukturen
        - SOCK_Stream zum Lesen und Schreiben auf einer Verbindung
        - SOCK_Acceptor für die Funktionen zum Initialisieren eines Sokket und zur Verbindungsannahme (Factory für SOCK_Stream-Objekte)
        - weitere Funktionen zum Erzeugen und Koordnieren der Threads
    - Die Klassen mit ihren Methoden sind kompakt und übersichtlich. Für unterschiedliche Plattformen können unterschiedliche Implementierungen genutzt werden. Ggf. ist auch Vererbung einsetzbar um die Systemspezialisierungen zu trennen.

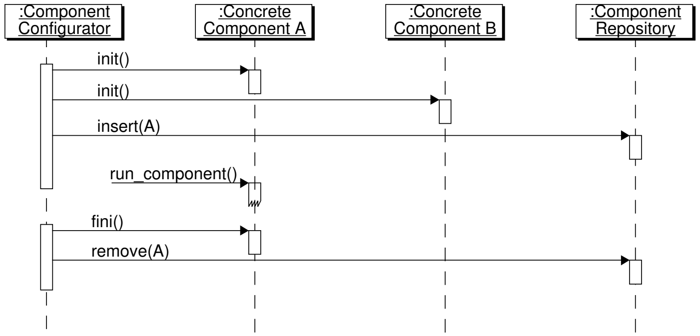
1 Service Access and Configuration Patterns (2)

- 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 (3)

- Interceptor
  - ➤ Erweiterung von Ausführungsumgebungen: dynamisches Hinzufügen von Diensten und automatische Benachrichtung 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)