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 wir d 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.
- Component Configurator (Service Configurator)
  - ➤ dynamisches Binden und Entfernen von Komponenten einer Anwendung
  - ➤ Realisierung über DLL und Austausch von Komponenten zur Laufzeit
    - Allgemeine Komponentenschnittstelle und tatsächliche Komponente werden getrennt.
      - über die allgem. Komponentenschnittstelle können tatsächliche Komponenten geladen, initialisiert und entfernt werden - unabhängig von der jeweiligen Funktionalität (Abstraktion von der DLL-Schnittstelle) .
      - die tatsächliche Komponente enthält Standardschnittstellen zur Konfiguration und die anwendungsspezifische Schnittstelle