Services

mit

Threads

Nach D. Lea, Kap. 6

Inhaltsübersicht zur Vorlesung

- Threads in Java, Sprachkonstrukte
- Modellierung von Threads mit FSPs
- Sicherheit
- Lebendigkeit
- Zustandsabhängige Aktivitäten
- Kontrolle von Nebenläufigkeit
  - Subclassing, Adapter, Meta-Objekte
- Threads als Anbieter von Diensten
- Architekturen auf Basis von Produzenten/Konsumenten
- Koordinierte Interaktion

Heute:
Services mit Threads

- Aufgabe: „execute server.operation(arguments) in new Thread“
- Varianten
  - Runnable Services
    - Übergabe von Parametern und Ergebnissen
  - Thread-Per-Message Proxies
    - komfortablere Schnittstelle als Runnable Services
  - Waiters und Composites
    - Überwachung der Terminierung
  - Early Replies
    - vorzeitiges Acknowledgement
  - Autonome Endlos-Schleifen
  - Polling Watches
- Terminierung
  - Join, Futures, Time-Outs, Completion Callbacks

Delegation einer Aufgabe an einen Thread zur asynchronen Bearbeitung

- „execute server.operation(arguments) in new Thread“
- Hindernis: Objekte, Methoden und Argumente lassen sich nicht einfach an Thread übermitteln
- Einzige Möglichkeit in Java: festdefinierte Methode run eines Runnable Objektes wird bei Thread.start() ausgeführt
  - run() hat keine Argumente
  - run() hat keine Rückgabewerte
- Grundfunktionalität von Java ist ausreichend
- Für typische Probleme sind jeweils Lösungsmuster/patterns bekannt
- Einordnung nach
  - Art des Aufrufs
  - Schnittstellen
  - Klassenstruktur

Art des Aufrufs

- Client kontrolliert Interaktion
  - Prozedural: Client initiiert Aufgabenbearbeitung und wartet Ende ab
  - Einweg: Client initiiert Aufgabenbearbeitung, keine weitere Kontrolle
  - Interaktiv: Client initiiert Aufgabenbearbeitung, erhält Ergebnisse
    - durch Terminierung des Threads
    - durch Nachfrage des Threads, führt ggfs zu Wartesituation
- Server kontrolliert Interaktion
  - Early Reply: Server liefert vor Bearbeitung vorzeitiges Acknowledgement
  - Prozedural: Server nimmt Aufgaben nur nach Erledigung vorheriger Aufgabe an.
  - Liefert Ergebnisse durch Rückruf (Callback) des Threads
  - agiert permanent in Endlosschleife im Hintergrund

Aufrufschnittstellen für Services

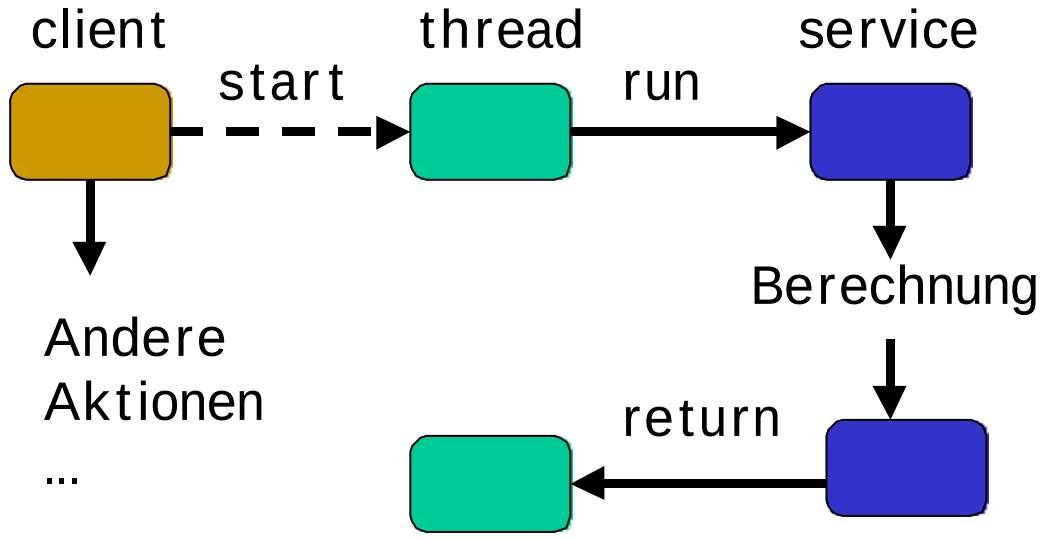
- Runnable:
  - Runnable Service, Dienst ist als run Methode eines Runnable Objektes verfügbar
  - Adapter, Dienst wird durch Adapter verfügbar
- Proxy, Protocol Adapter:
  - Client Aufrufe habe ähnliche Syntax wie prozedurale Methodenaufrufe
  - prinzipieller Unterschied: weil Serviceaufruf für Client nicht blockierend sein soll, erfolgt Ergebnisbereitstellung unabhängig vom return des Aufrufs.

Klassenhierarchie

- Serviceklasse muß Runnable implementieren
- 3 mögliche Varianten
  - aus Thread abgeleitete Klasse
    - nutzlos, keine ererbte Zusatzfunktionalität
    - jede Methode kann den aktuellen Thread mittels Thread.currentThread() erhalten
  - aus anderer Klasse abgeleitet
    - häufige, einfache Variante
    - ererbte, öffentliche Methoden/Variable können Interferenz bewirken
  - eigenständige Klasse
    - als Adapter oder mit eigenem Code zur Diensterbringung

Commands - Services ohne Interaktion

- Runnable Service Pattern
  - Codierung einer einzelnen Methode in einer eigenständigen Service Klasse
  - Client ruft nicht direkt Methode auf, sondern startet Methode innerhalb eines Threads
  - Variante 1: Basisfunktionalität wird in der Service Klasse codiert
  - Variante 2: Service Klasse als Adapter, Delegation des Methodenaufrufs and bestehende Klasse (Runnable Adapter)



Runnable Services

- Run() erlaubt keine Parameter
- Ausweg:
  - Runnable Object hat Instanzvariable für Parameter
  - Konstruktor reicht Parameterwerte an Instanzvariable durch
- Vorteilhaft: One-Shot-Design
  - Parameter werden nur einmal gesetzt, sind unveränderlich (immutable)
  - run() wird nur einmal oder stets mit denselben Parametern aufgerufen

    Achtung: je Thread nur einmal start(), aber runnable Object kann mehrfach zur Instanziierung eines Threads genutzt werden!
  - Einfache Vermeidung von Interferenz und Zustandskonflikten, keine Synchronisation erforderlich
- bei mehrfacher Verwendung von veränderlichen Runnable Objects, mittels Synchronisation Zuweisung von Parameterwerten sicherstellen, ggfs darüberhinaus passende Identität des Threads abprüfen. Ziel jede Instanz rechnet mit „ihren“ Parameterwerten.

Runnable Services

- Aktivierung kann beim Client oder Server liegen
- Sei RO runnable Object und param die Parameter des Services
- Clientseitige Aktivierung: new Thread(new RO(param)).start() ;

  Client instantiiert RO mit den Parametern des Services und startet neuen Thread mit Parameter RO
- Serverseitige Aktivierung: new Thread(this).start() ;

  wobei der Server=RO durch den Client erzeugt wird, ONE-SHOT-DESIGN

  Achtung: Mißbrauchsschutz wg möglicher Mehrfachverwendung von run()

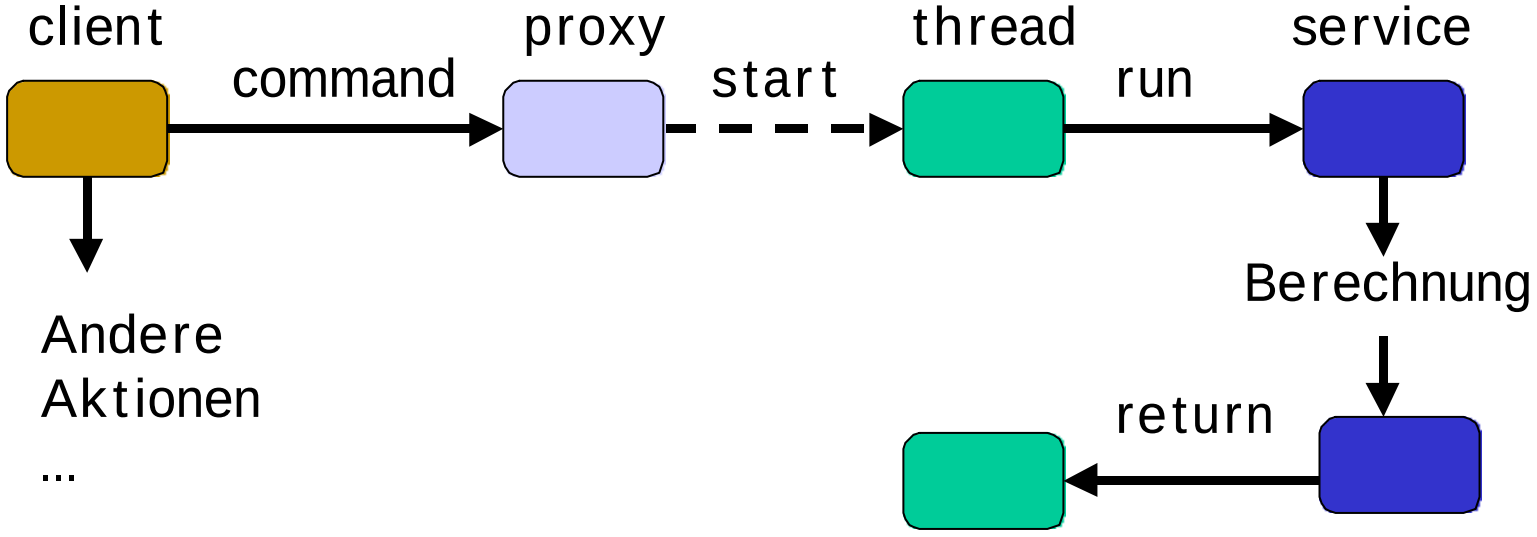
```
Public class AutoStartRO implements Runnable {
protected Thread me_ ;  protected Object p_ ;
AutoStartRO(Object param) { p_ = param ;
   me_ = new Thread(this) ; me_.setDaemon(true) ; me_.start() ; }
public void run() {
   if ( Thread.currentThread() ==  me_ ) service() ;
}}
```

Daemon Thread lebt nicht länger als alle Anwendungsthreads

Policy: Inaction

Thread-Per-Message Proxies und Gateways

- Design erlaubt gleichartige Aufrufe für Implementierungen mit Threads wie bei entsprechenden herkömmlichen Klassen
- 2-stufige Struktur, vereinheitlicht Design und Nutzung
- erlaubt zusätzlich Prioritätskontrolle und ThreadGroups zu verwalten
- Proxy: verwaltet nur 1 Art von Service, Gateway: verschiedene Services
- Gateways sind Kompromiß zu Client- und Serverkontrollierter Aktivierung
- Gateways ähnlich zu Factory Pattern, dort jedoch Rückgabe eines Handles (ebenfalls möglich, bewirkt Clientkontrolle des Threads)



Thread-Per-Message Proxies und Gateways

- Proxies vereinfachen Threadnutzung bei Parametern und Rückgabewerten.
- Proxies entweder speziell für eine Klasse kodiert, ... oder allgemeine Lösung ?
- Standard Interfaces
  - Runnable: Parameterlose Methoden ohne Rückgabewerte
  - weitere denkbar: 1 Parameter, 2 Parameter, ...
  - Typische Vertreter
    - Anwendung einer Operation

      public interface Procedure { public void apply(Object x) ; }
    - Erzeugung, Rückgabe eines Objektes

      public interface Generator { public Object next() ; }
    - Anwendung einer Operation mit Rückgabe des Resultates

      public interface Function { public Object map(Object x) ; }
    - Testen einer Eigenschaft

      public interface Predicate { public boolean test(Object x) ; }
    - Anwenden einer binären Operation mit Rückgabe des Resultates

      public interface Combiner { public Object combine(Object x, Object y) ; }

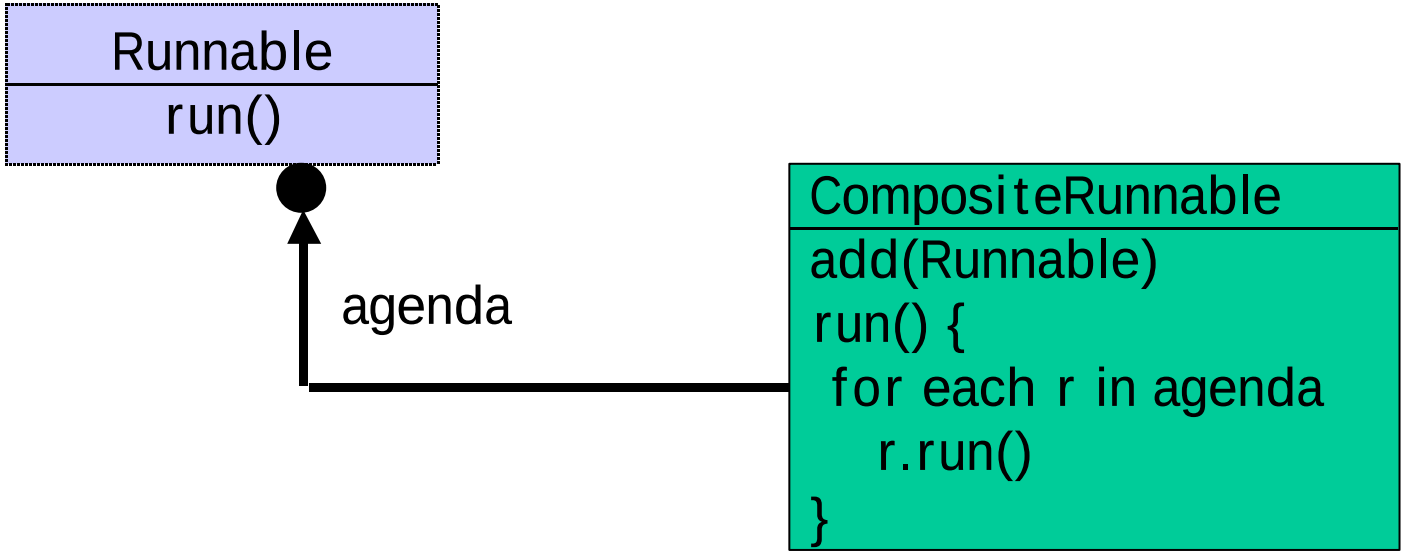
Waiter - ein einfacher Runnable Adapter

- Runnable Adapter realisieren asynchrone Nachrichtenübermittlung innerhalb eines Java Programs.
- Simple Variante: Waiter (nein nicht Kellner, sondern ...)

  Adapter, der einen Service innerhalb eines Threads erbringt und die Terminierung des Services abwartet (warten -> waiter), wobei der Dienst von der Art „oneway send-and-forget“ ist, d.h. einmalig aufgerufen wird und keine weitere Ablaufkontrolle erfolgt
- entspricht Runnable Service mit Delegation
- Schema:
  - Instanzvariable mit Referenz auf Zielobjekt und weitere für Argumente der aufzurufenden Methode
  - ggfs Instanzvariable und Get/Set Methoden zur Sicherung von Rückgabewerten
  - Konstrukturmethode zur Initialisierung der Instanzvariable
  - run() zum Aufruf der Methode des Zielobjektes und zur Sicherung von Rückgabewerten
  - für die einmalige Nutzung des Waiters, synchronized run() und ggfs Thread ID kontrollieren

Composites - Runnable Adapter für mehrere Services

- Idee: Adapter hat interne Liste (Agenda) mit Aufgaben, die in run() abgearbeitet werden und die erst zur Laufzeit bekannt sind.
  - Liste kann dynamisch ergänzt werden add(Runnable r)
  - sequentielle Abarbeitung innerhalb des eigenen Threads
  - nicht-sequentielle Abarbeitung durch eigens geschaffene Threads:
    - dann intern jedoch baum/graphartige Strukturen für Verzweigungen, Reihenfolgen, Loops etc zur Beschreibung der Ablaufs



Early Replies

- Idee: vor Diensterbringung wird vom Server bereits ein Resultat zurückgegeben
- Widerspruch in sich ? -> Resultat ist meist Acknowledgement
- wird bei asynchronen Hintergrundthreads verwendet, die gelegentlich getriggert werden
- passt gut zu Antwortverhalten bei Methodenaufrufen
- bei Reply wandert einerseits Kontrolle an den Client zurück und gleichzeitig verbleibt ein Kontrollfaden zur asynchronen Bearbeitung beim Server
- Nutzen: zur Bestätigung der Auftragsannahme/Auftragseingangssequenz bei Services, die keine Rückgabewerte haben und deren Bearbeitung zeitintensiv ist.
- WARNUNG: early replies kann zu subtilen Verwechselungen führen: Return bedeutet nicht, daß der Dienst vollständig erbracht wurde (übliche Return Semantik) sondern, daß der Auftrag eingegangen ist. Dies kann sich bei der Synchronisation von parallelen Abläufen bemerkbar machen.

Early Replies in Java

- Lediglich durch Verhalten von start() der Thread Klasse unterstützt.
- Dies reicht aus!
- Schema:
  - Instanzvariable für das Service Objekt und für die Parameter der Servicemethode
  - Konstruktor zur Initialisierung der Instanzvariable
  - run() Methode, die mit Hilfe der Methoden des Service Objektes den geforderten Dienst erbringt.
  - Sonderfall: Service ist parameterlos und hat keine Rückgabewerte, dann ist Hilfsklasse überflüssig und Klasse des Service Objektes kann bereits Runnable implementieren.

Beispiel Code: Early Reply bei triggerActivity

```
Class ServerActivity implements Runnable {
private ERServer s_ ; private int j_ ;
ServerActivity(ERServer s, int j) { s_ = s  ; j_ = j ; }
public void run() { s_.doActivities(j_) ; }

public class ERServer {
public synchronized int triggerActivity(int j) {
  int result = computeResult() ;
  new Thread(new ServerActivity(this,j)).start() ;
  return result ;
}
synchronized void doActivities(int j) { activity1() ; activity2(j) ; }

protected int computeResult(){ ... }
protected synchronized void activity1(){ ... }
protected synchronized void activity2(int j){ ... }
}
```

Nicht-terminierende Hintergrundthreads

- Erfordern Endlosschleifen for(;;) { doService() ; }
- Anwendungsbeispiele
  - Autonome Objekte, die bei Instantiierung Thread erzeugen und starten, durch den in einem Update Loop unabhängig über die Zeit Updates am Objekt durchgeführt werden.
  - Kontinuierliche Animationsloops in Applets
  - Simulationen von Zufallsprozessen
  - Event Loops und Listener
  - Checkpointing Mechanismen, die periodisch Sicherheitskopien ziehen und persistent abspeichern.
- Implementierung mit gleichen Mechanismen zur Parameterübergabe und zur Delegation von Servicemethodenaufrufen wie bisher.

Asynchrone Hintergrundthreads: Polling Watches

- Dämon Thread mit Endlosschleife, der gelegentlich Bedingungen prüft und abhängig davon Aktivitäten durchführt
- eine und häufig verwendet Hilfsklasse
- Polling = Abfrage eines Zustands
- Vorsicht: Polling impliziert Busy Waiting mit üblichem Risiko für Performance!
- Anwendbar, wenn
  - Verpassen von Gelegenheiten tolerierbar
  - zeitnahes Durchführen der Aktivität bei Eintreten der Bedingung nicht gefordert
  - Fehlalarme tolerierbar sind oder verhindert werden können. Fehlalarm: Aktivität wird durchgeführt, obwohl die Bedingung inzwischen nicht mehr gilt. Verhinderungsstrategie (wie üblich): definiere Aktivität als synchronized mit Balking
  - Polling Overhead ist tolerierbar, weil Abfragen nicht zu häufig erfolgen.

Asynchrone Hintergrundthreads: Polling Watches

- Design Schritte
  - Instanzvariable für alle Teilnehmer, Parameterwerte für Methodenaufrufe zur Bedingungsüberprüfung und für Aktivitäten, Initialisierung wie üblich im Konstruktor
  - „checkAndAct“ Methode, die Bedingungen prüft und ggfs Aktivitäten durchführt
  - Run() Methode, die fortlaufend checkAndAct und sleep im Wechsel aufruft. Run() kann bei Erfolg entweder terminieren und in einer Endlosschleife fortfahren.
  - Konstruktor zur Initiierung des Threads.
- Kombination mit Composites Pattern
  - Behandlung mehrerer Fälle durch einzelne CheckAndAct Objekte, die in einer Agenda verwaltet werden.
  - Möglich: Abarbeitung der Agenda nach Prioritäten
  - Möglich: Optimierung durch Abhängigkeiten zwischen Bedingungen.

Rückgabe von Ergebnissen bei Terminierung

- Problem: run() hat keine Parameter und keinen Rückgabewert
- bisher: Parameterübergabe bei Konstruktoraufruf des Runnable Objektes und Speicherung in Instanzvariable
- Speicherung von Rückgabewerten, Ergebnissen in Instanzvariable bei Servicemethoden natürlich ebenfalls möglich
- Übermittlung an Client ?
  - Durch Terminierung des Thread und Get Methodenaufrufe des Clients
    - Abfrage von Thread.isAlive() oder Latches für Instanzvariable des Runnable Objektes: NACHTEIL Busy Waiting !!!
    - Besser Terminierungserkennung mit Join
  - Futures
  - Time-Outs
  - Rückrufe (Callbacks)

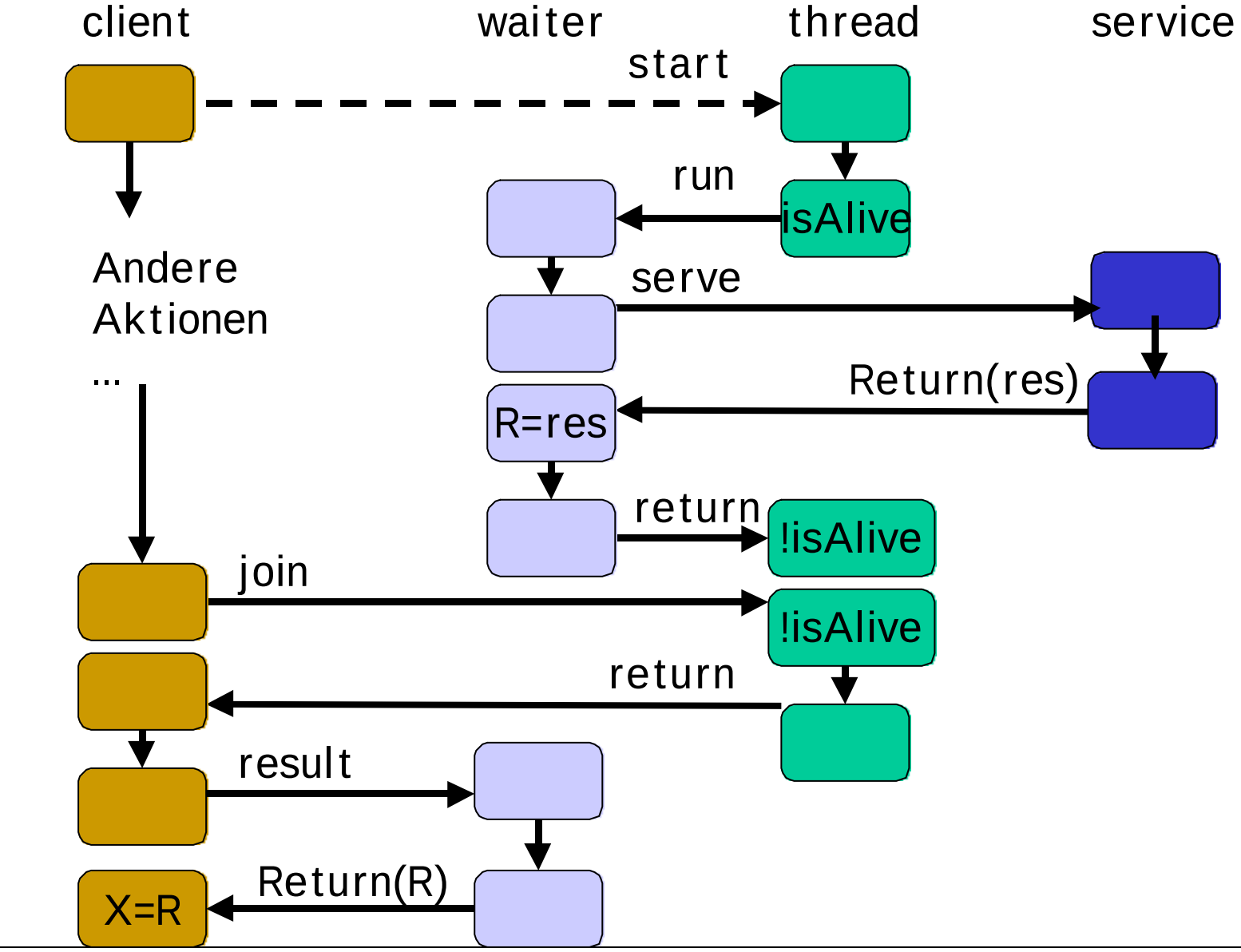
Ergänzung des Waiter Patterns um Join Mechanismus

- Ergebnisse der Servicemethode werden in Instanzvariable result des Runnable Objektes gehalten.
- Client kennt Thread und Runnable Object, daher Interaktion wird vom Client kontrolliert:

```
waiterThread.join() ;
result = waiter.getResult() ;
```

- anwendbar, wenn nur 1 Thread gestartet wird und das Warten auf Terminierung innerhalb derselben Client Methode erfolgt.
- Alternativen: Completion Callbacks und Group Proxies

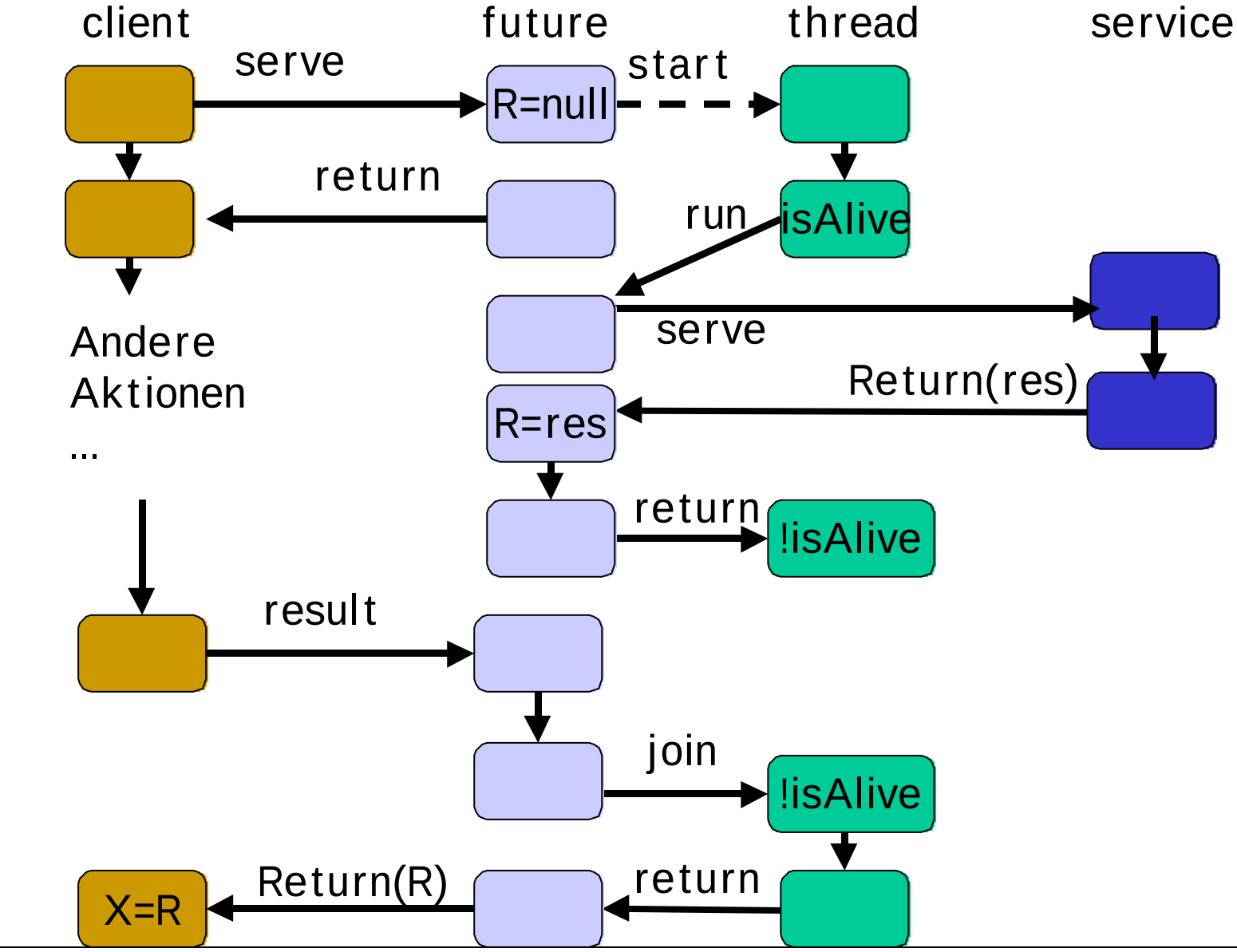
Beispielablauf:



Futures = Proxis mit „wait-by-necessity“ für Ergebnisse

- Transparentere Nutzung von Join bei geringerer Flexibilität
- Grundidee: Proxy kapselt Threadnutzung, Client muß bei Nutzung der Ergebnisse ggfs auf Terminierung des Service Threads warten
- einfache Nutzung von Methoden des Futures durch den Client, Unterschied zu Methoden ohne Threads liegt lediglich in der Abtrennung einer Ergebnisrückgabe durch eigene Abfragemethoden.
- Kontrolle der Interaktion liegt in diesem Fall beim Server=Future

Beispielablauf:



Time-Outs

- Join Methode erlaubt Implementierung mit Timeouts

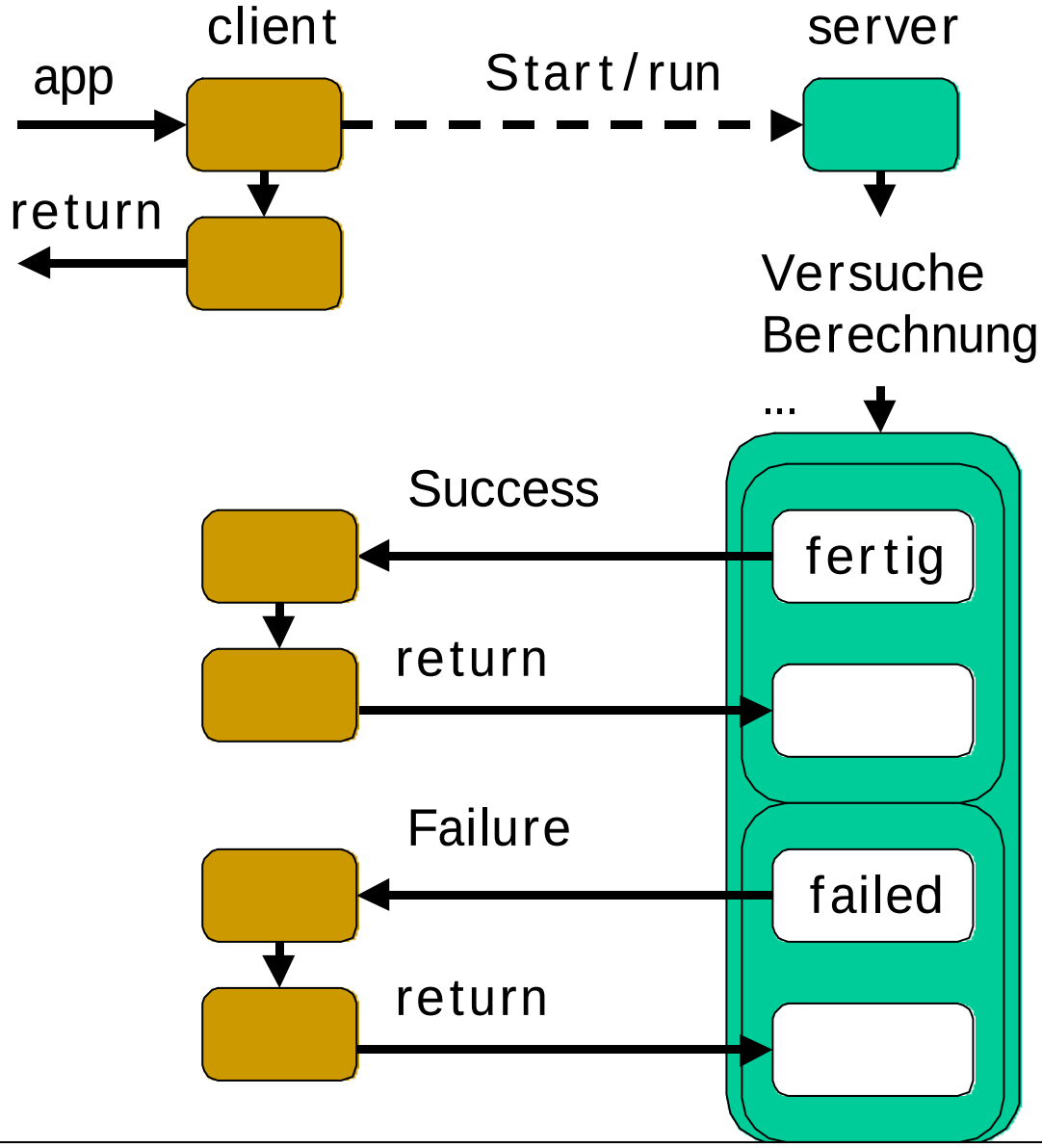
  Thread.join(long milliseconds)
- als Reaktion kann Signal an Thread zum vorzeitigen Abbruch erfolgen
  - bei D. Lea noch veraltete Fassung mit Thread.stop()
  - besser: in Runnable Objekt Flag setzen, das in run() gelegentlich geprüft wird. Erlaubt geordnete Terminierung unter der Kontrolle von run().

```
waiterThread.join(timeOutValue) ;
if (waiterThread.isAlive())
  waiterThread.setStop() ;
else
   result = waiter.getResult() ;
```

Completion Callbacks

- Completion Callbacks sind Methoden, die der Client zur Verfügung stellt, damit der Server nach Berechnung der Resultate, diese beim Client abgeben kann.
- Interaktionskontrolle liegt beim Server
- Pattern ist etwas flexibler als Join Variante aber schwerer zu kontrollieren.
- Anwendbar, wenn
  - eine/mehrere Aktionen bei Beendigung der Berechnung durchgeführt werden müssen (in Ergänzung zur Ergebnisrückgabe)
  - die Rückgabe Aktionen müssen nicht in einen speziellen Aufrufkontext eingebunden sein und können als eigenständige Methoden formuliert werden
  - Beendigung der Berechnung (logische Terminierung) fällt nicht mit der Terminierung des Threads (der run Methode) zusammen. Beispiel: Server betreibt Early Reply und Thread-Per-Message Pattern, so daß Serverterminierung nichts über Terminierung des Servicethreads aussagt.
  - Objekte, die Callback Methode ausführen, müssen Initiator nicht kennen.
  - Kontext asynchrone, nachrichtenbasierte Kommunikation zuläßt.

Beispielablauf:



Design Schritte bei Callbacks

- Festlegen der relevanten Resultatmöglichkeiten, typisch:
  - erfolgreiche Berechnung von Ergebnissen (Success) und
  - Abbruch der Berechnung (Failure)
- Definieren von Interfaces für die Callback Methoden
- Erweiterung des Konstruktors der Runnable Klasse um Referenzen auf Objekte, die Callbackmethoden zur Verfügung stellen und Erweiterung der Run Methode, so daß Callbackmethoden passend aufgerufen werden. Achtung: Referenzen dürfen NULL sein!
- (Optional) zusätzlicher Konstruktor für Version ohne Callbacks.
- Erweitere Clients um Callback Behandlung oder erzeuge eigenes Handler Objekt für Callbacks, das an den Server übermittelt wird.

- Zusätzliche Varianten:
  - Rückgabe weiterer Information für Failures
  - Bewachte Callback Methoden, die auf den Zustands des Clients Rücksicht nehmen, z.B. Reihenfolge bei mehreren Services, ...

Zusammenfassung: Services mit Threads

- Basisfunktionalität:
  - run() für parameterlose, asynchrone Aufgaben ohne Rückgabewert
  - start() für Early Reply Schema
- darauf aufbauend eine Reihe von Pattern, jeweils
  - Parameterübergabe bei Konstruktoraufruf des Runnable Objektes
  - Ergebnisrückgabe
    - aktiv durch Client: Polling, join, wait/notify
    - aktiv durch Server: Callback, bewachte CallbackS
- Patterns
  - Futures, Joins, Polling Watches, Loops, Early Replies, Composites, Waiters, Thread-Per-Message Proxies, Runnable Services