Zustandsabhängige Aktionen

Nach D. Lea, Kap. 4

Inhaltsübersicht zur Vorlesung

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

Heute: zustandsabhängige Aktionen

- Beobachtung: manche Aktionen, Methoden lassen sich nur in bestimmten Zuständen durchführen
- Zustand läßt sich interpretieren als
  - die Erfüllung von Vorbedingungen
    - z.B. bzgl der Werte von Variablen
  - die Verfügbarkeit von Ressourcen
- geeignete Sichtweise: Thread ist Automat, der durch Aktionen seinen Zustand wechselt und zustandsabhängig Aktionen durchführen kann.
- Wie läßt sich mit zustandsabhängigen Aktionen verfahren ?
  - Unerwünschter Effekt: Thread ist nicht in der Lage Aktion durchzuführen, wie soll sich der Thread dann verhalten ?
  - Verhaltensmuster werden als Policies bezeichnet und im folgenden genauer betrachtet.

Policies zur Berücksichtigung von Vorbedingungen

- Unbedingte Aktionen: Vorbedingungen wirken nur als Disclaimer und werden ignoriert, Aktionen werden ungeachtet der Konsequenzen betrieben

echte Policies:

Check&Act, Konservative, Pessimistische Strategien:

- Inaction: keine Reaktion, falls Vorbedingung nicht erfüllt
- Balking: Rückgabe von Fehlermeldungen, falls Vorbedingung nicht erfüllt
- Guarded Suspension: die Ausführung der Aktion wird verzögert, bis die Vorbedingung erfüllt wird

Try&see, Optimistische Strategien:

- Provisional Action: die Durchführung wird vorgegeben, die Resultate allerdings nicht bestätigt, bis Erfolg sichergestellt ist
- Rollback/Recovery: Aktion wird versuchsweise durchgeführt, bei Fehlverhalten erfolgt Recovery/Rollback bzgl aller Effekte

  (Rollback bis zum Startpunkt, Recovery bis zum letzten Sicherungspunkt)
- Retry: wiederholte, versuchsweise Durchführung von Aktionen bei Mißerfolg

Diskussion

- Check&Act führt zu relativ einfachen und tendentiell auch zuverlässigen Designs bei Multithreading.
- In verteilten Applikationen jedoch dann kritisch, wenn die Prüfung von Vorbedingungen nicht die Durchführbarkeit von entfernten Aktionen sichern kann.
- Keine dieser Strategien ist perfekt, optimal!

- Auswahl einer Strategie findet in früher Design Phase statt mit entsprechenden Folgen.
- Dazu eine Reihe von Randüberlegungen
  - Sind Vorbedingungen intern berechenbar? Ist die Durchführbarkeit einer Aktion lokal kontrollierbar?
  - Sind Vorbedingungen extern berechenbar? Kann der Nutzer vorab feststellen, ob Objekt Aktion erbringen kann? Ferner Gültigkeit der Aussage zwischen Feststellung und Durchführung der Aktion ?

Weitere Aspekte

- Wie aufwendig ist die Berechnung von Vorbedingungen?
- Ist die Anfrage überhaupt formal zulässig, z.B. bzgl der Rechte ? Falls nicht, ist eine Überprüfung der Vorbedingungen irrelevant ...
- Hat die Überprüfung von Vorbedingungen unerwünschte Seiteneffekte ?
- Sind die Vorbedingungen durch das Host Objekt erfüllbar ? Die Verantwortung zur Herbeiführung von Vorbedingungen kann beim Host oder beim Client liegen.
- Liegt eine Wartesituation beim Zugriff auf eine geteilte Ressource vor ?
- Ist eine unbestimmte Wartezeit auf das Eintreffen der Vorbedingung für den Client akzeptabel ? (z.B. bei Now-or-Never Semantik nicht)
- Kann Warten auf Eintreffen der Vorbedingung sinnvoll sein ?
- Ist Failure für den Client akzeptabel ?
- Kann der Host den Mißerfolg überhaupt selbst feststellen ?

Weitere Aspekte

- Sind alternative Aktionen, Methoden verfügbar ?
- Sind Fehlalarme aufgrund nur geschätzter Vorbedingungen akzeptabel für den Client ?
- Lassen sich Seiteneffekte von fehlgeschlagenen Aufrufen rückgängig machen?
- Müssen Clients spezielle Aktionen durchführen, um Fehlschläge zu verkraften?
- Lassen sich bei Fehlversuchen Aktionen von innen (Host) oder von außen (Client) versuchsweise wiederholen ?

Diese Aspekte sollten bei der Auswahl einer Strategie
- pessimistisch: Inaction, Balking, Guarded Suspension
- optimistisch: Provisional Action, Rollback/Recovery, Retry

für eine konkrete Anwendung berücksichtigt werden.

Repräsentation von Zustandsinformation
Interfaces

- Interfaces können als semi-formales Gerüst dienen
- Interfaces können keinen Bezug auf Instanzvariable oder Code nehmen

=> Beschreibung erfordert Angabe von Invarianten und Funktionalität auf eine Art und Weise, dass Code nicht gelesen werden muss

Beispiel: Zähler mit beschränktem Wertebereich

```
public interface BoundedCounter {
public static final long MIN = 0 ;   // minimaler zulässiger Wert
public static final long MAX = 10 ; // maximaler zulässiger Wert

public long value() ; // invariant: MIN <= value() <= MAX
                      // initial: value() == MIN
public void inc() ;   // increment nur bei value() < MAX
public void dec() ;  // decrement nur bei value() > MIN
}
```

Repräsentation von Zustandsinformation
Logischer Zustand vs Realer Zustand

- Realer Zustand: aktuelle Wertebelegung aller Variable einer Instanz
- dadurch können extrem große Mengen an möglichen realen Zuständen entstehen (Kreuzprodukt der Wertebereiche)
- für die Durchführbarkeit von n Methoden sind jedoch nur maximal 2^^n Zustände erforderlich
- Beispiel: Zähler mit beschränktem Wertebereich
  - MAX-MIN reale Zustände, aber nur 3 logische Zustände

Zustand	Bedingung	Inc	Dec
Top	Value == MAX	N	Y
Middle	MIN<value<MAX	Y	Y
Bottom	Value== MIN	Y	N

Repräsentation von Zustandsinformation
Logische Zustände

- Logische Zustände werden
  - als Prädikate, als boolesche Ausdrücke definiert
    - eigenständige interne boolesche Methoden
    - einfache boolesche Bedingungen im Code
  - durch explizite Zustandsvariable definiert
    - erfordert jeweils Update bei Änderungen (Overhead)
    - erleichtern die Erstellung von Traces, Debugging
    - Spezialfall: Rollenvariable halten nach, welche Rolle ein Objekt aktuell einnimmt (Produzent/Konsument) und Verzögern Methodenausführung für Rollen, die nicht aktuell sind
    - Implementierungssalternative: eigenes Zustandsobjekt (States as Objects Pattern)
- Wenn die Analyse von Zuständen einer Klasse zu komplex wird helfen
  - State charts/Zustandsdiagramme, Aktivitätsdiagramme (UML)
  - Tabellen, Entscheidungsbäume, Automaten, ...

Repräsentation von Zustandsinformation Historienvariable

- Einfache Zustandsinformation ist manchmal unzureichend.
- Alternative:

  History Log protokolliert alle Nachrichten (Eingang&Ausgang) + Aktionen
- Beispiel:
  - Methode wird nur dann angesteuert, wenn innerhalb der letzten k Nachrichten, dieselbe Anforderung m-mal aufgetreten ist.
- Aus dieser Perspektive erscheinen einfache Zustandsvariable als Zusammenfassung von Historienvariable.
- Auch: Execution State Variable/Aktive State Variable
  - Beispiel: Nachrichtenzähler
- Implementierungshinweis: Klassen sollten für Zustandsvariable protected Accessor Methoden beinhalten, damit Vererbung einfacher Zugriff möglich ist.

1. Konservative Policy
Guarded Suspension - bewachte, verzögerte Ausführung
Vorüberlegungen

- sequentieller Fall: if-then-else,

  Entscheidung wird ad-hoc gefällt, Warten wäre unsinnig

- Multithreading:

  Eintreten einer Bedingung kann sehr wohl durch andere Threads bewirkt werden, Warten kann sinnvoll sein

Ungünstige Reihenfolge von Nachrichten kann durch Scheduling verursacht sein!

Sichtweise:

Guarded Suspension agiert unter der Annahme,

- dass die für die Ausführung einer Aktion erforderlichen Zustandsänderungen irgendwann eintreten werden (Lebendigkeitsannahme),
- oder dass bei Ausbleiben der Zustandsänderungen, die Aktion besser beliebig verzögert wird, d.h. nicht durchgeführt wird (Stillstand).

Guarded Suspension - bewachte, verzögerte Ausführung

- Schafft lokale Scheduler für ein Objekt
- generalisiert damit Lock-Mechanismus (synchronized)
- Beispiel: PSEUDO-CODE für Zähler mit beschränktem Wertebereich

```
Pseudoclass BC implements BoundedCounter {
protected long c = MIN ;

WHEN (true) ACCEPT public long value() { return c ; }

WHEN (c < MAX) ACCEPT public void inc() { ++c ; }

WHEN (c > MIN) ACCEPT public void dec() { --c ; }
}
```

Guarded Suspension - Implementierung

- Basiert auf Kopplung aus wait() und notifyAll() Methoden
- Schema
  - erzeuge für jede Bedingung, auf deren Eintritt gewartet werden muß, eine bewachte WAIT - Schleife
  - rufe NotifyALL in jeder Methode auf, die Zustandsänderungen durchführt, so daß eine Warte-Bedingung erfüllt sein könnte
- unterschiedliche Realisierungsvarianten
  - Wait und Notify, Interrupt
  - Busy-Wait
  - Notifications: jeweils an alle oder nur einzelne Threads

    (Aufwand beachten)

Guarded Suspension
Implementierung mittels WAIT (und notify(), interrupt())

- Typisches Muster aus while-Schleife mit Bedingungsabfrage
- Z.B. in eigener Methode gekapselt

```
...
Protected synchronized void awaitCond() {
  while (!cond) {      // cond hier als Instanzvariable verfügbar
     try { wait() ; }
     catch (InterruptedException ex) {}
  }
}
```

- wesentlich
  - Bedingung in while Schleife prüfen, damit Bedingung bei Beendigung des Wartens auch wirklich gilt. Problem z.B. aus unpassendem notifyAll oder Zustandsänderungen zwischen notify und Ausführung
  - Auffangen der IntteruptedException wg Unterbrechung durch interrupt()
- Achtung: bewachte Methode muß bei awaitCond() selbst in konsistentem Zustand sein! => typischerweise Bedingungen direkt bei Aufruf abprüfen

Guarded Suspension
Implementierung mittels Busy Waiting (UNGÜNSTIG)

- SPIN loops sind ungünstig

```
...
Protected void spinWaitCond() {
  while (!cond) {
    Thread.currentThread().yield() ;
  }
```

- Vergeudung von CPU-Zyklen
  - bei Verwendung von yield() Fehlermöglichkeit: werden für die Threads Prioritäten vergeben und hat der wartende Thread eine höhere Priorität als der Thread, der die Bedingung herstellen soll, verhindert Busy-Waiting, dass die Bedingung jemals erfüllt wird.
  - Durch Scheduling können Phasen, in denen Bedingung erfüllt ist, verpaßt werden!
  - Fast unmöglich, synchronized bei Busy Waiting sinnvoll zu setzen!
  - Instanzvariable cond müssen als volatile definiert werden.
- Vorherige Konstruktion mit wait ist deutlich besser !!!

Guarded Suspension
Notifications zur Aufhebung der Wartesituation

- Benachrichtigungen können/sollen jeweils ausgelöst werden, wenn Werte von Variable für logische Zustände geändert werden.

=> Kapselung der Zugriffe auf diese Variable, innerhalb der Zugriffsfunktionen wird dann jeweils notifyAll() aufgerufen.

- Dies vermeidet häufigen Fehler, dass bei Manipulationen an Zustandsvariablen Benachrichtigungen vergessen werden.
- Diese Technik sollte initial eingesetzt werden, bei Performance Problemen wg zu häufiger Benachrichtigungen, sollte anschließend selektiv das Nachrichtenaufkommen verringert werden.
- Benachrichtigungen sollten üblicherweise am Ende einer Methode ausgelöst werden.
  - Dies vermeidet Überlegungen, welcher Thread sinnvollerweise als erstes weiterzuführen sei.
  - Falls dies nicht möglich ist, sollten alle synchronisierten Blöcke verlassen werden und anschließend an das notifyAll() ein yield() durchgeführt werden.

Guarded Suspension
Implementierung mittels Single Notifications

- Beispiel: Zähler mit beschränktem Wertebereich erfordert notifyAll(), weil Threads auf theoretisch auf 2 unterschiedliche Bedingungen warten können.
- Ein einzelnes notify ist ausreichend, wenn
  - alle Threads, die auf das Eintreffen von Benachrichtigungen dieses Objektes warten, auf die gleiche Bedingung warten und
  - jede Benachrichtigung maximal einen Thread befähigen kann fortzufahren
- z.B. bei einer binären Semaphore
- ein einzelnes notify wird schneller abgearbeitet als ein notifyAll
- notify kann kaskadierend implementiert werden
  - falls benachrichtigter Thread selbst doch nicht fortfahren kann wird, löst er weiteres notify aus.
  - führt im Grenzfall zu notifyAll
  - Effizienz längerer Kaskade im Vergleich zu notifyAll zweifelhaft

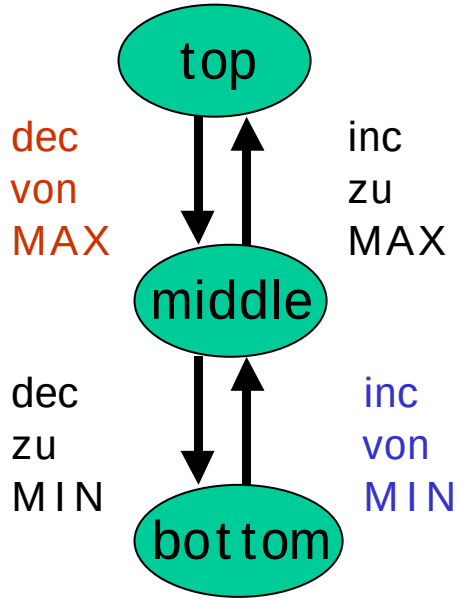
Zustandsvariable zur Reduktion von Benachrichtigungen

- Genauere Betrachtung von Zustandsvariable und der Wirkung von Zustandsänderungen auf wartende Threads kann lohnend sein!
- Beispiel: Zähler mit eingeschränktem Wertebereich

  Hier sind nur Decrement bei MAX und Increment bei MIN

  relevant, d.h. nur sie können wartende Threads aktivieren!

```
...
public synchronized void inc() {
  while (c== MAX)
    try { wait();} catch (InterruptedExeceptin ex) {}
  if (c++ == MIN) notifyAll() ;
}
public synchronized void dec() {
  while (c== MIN)
    try { wait();} catch (InterruptedException ex){}
  if (c--==MAX) notifyAll() ;
}
```



Beispiel

- Bisheriges Beispiel eines Zählers ist lediglich illustrativ
- Sinnvollere Anwendung der Konzepte:

  Realisierung eines beschränkten Puffers

  Grundidee: zyklischen Puffer auf Vektor/Array fester Länge implementieren

  Guarded Suspension macht z.B. im Kontext eines Produzenten/Konsumenten-Verhältnisses Sinn
- Aufgabe: Implementiere Puffer für Interface mit Guarded Suspension

```
Public interface BoundedBuffer {
public int capacity() ; // Invariante 0 <= capacity
public int count() ;     // Invariante 0 <= count <= capacity
public void put(Object x) ; // einfügen falls count < capacity
public Object take() ; // entferne falls 0 < count
}
```

Kapselung von Zustandsvariablen

- Implementierung mit expliziten Zustandsvariablen manchmal vorteilhaft
- Zustandsvariable sollten dann gekapselt werden
  - get und set Methoden
  - nach Update kann mit Methode checkState
    - der jeweilige Zustand geprüft werden und ggfs Benachrichtigungen ausgelöst werden
    - simple Variante: bei allen Änderungen notifyAll
    - aber auch hier lokal bessere Variante, die nur auf relevante Änderungen des lokalen Zustands reagieren.
  - Bei Vererbung kann checkState dann überlagert werden und spezialisiert werden.

Ein bekannter Spezialfall: Latches

- Ein Latch (dt: Schnappschloß) ist eine Variable, die nach ihrer Initialisierung nur noch ein einziges Mal gesetzt wird und dann nicht mehr verändert wird
- in Analogie: das Schloß ist zugeschnappt und bleibt dann geschlossen
- Anwendungsmöglichkeiten
  - Instanzvariable mit Referenz zu anderem Objekt, z.B. managed ownership von Ressourcen, wobei Ressource nur einmal zugewiesen wird (single static assignment)
  - Terminierungsanzeige: Wert ein booleschen Variable wechselt false->true um anzuzeigen, dass Dienst vollständig erbracht ist:
  - Erreichen einer Zeitgrenze: einmal erreicht, wird die Zeit nie wieder verfügbar.
  - Eintreten eines Ereignisses, Eintreffen einer Nachricht, eines Signals
  - Eintreten spezieller globaler Zustände, z.B. Thread ist terminiert, Deadlock ist eingetreten, alle Referenzen auf ein Objekt sind verloren gegangen.

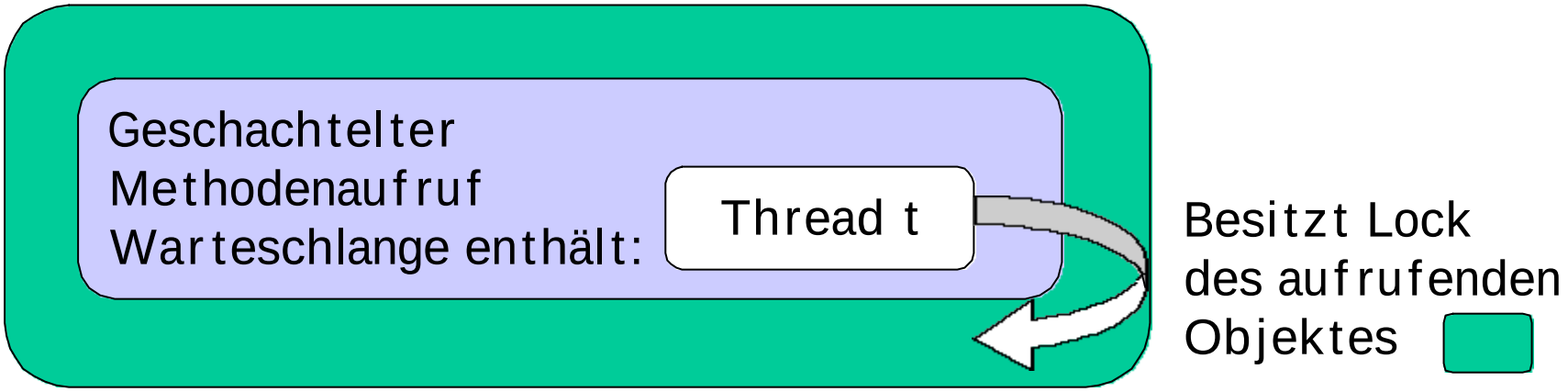
Beispiel: Initiierung eines aufwändigen Hilfsobjektes durch einen eigenen Thread Hinit

```
class Helper { ... void help() { ... } }
class Hinit implements Runnable { protected Hserver s_ ;
Hinit(Hserver s) { s_ = s ; }
public void run() { Helper h = new Helper() ; s_.initial(h) ;}
}
```

```
public class Hserver { protected Helper h_ ; // latch
public Hserver() { h_ = null ; new Thread(new Hinit(this)).start() ;}
synchronized void initial(Helper h) {
  if (h_ == null) { h_ = h ; notifyAll() ; }
}
protected Helper helper() {
  if (h_ == null)
    synchronized (this) {
      while (h_==null) try { wait() ; } catch (InterruptedExecption ex) {}
    }}
   return h_ ; }
public void delegatedAction() { helper().help() ; } }
```

Nested Monitor Problem / Lockout Problem

- Bei der Verwendung von Synchronisation folgendes Problem auftreten



- Schachtelung von synchronized Methoden unterschiedlicher Objekte kann zu Deadlocks führen, insbesondere bei enthaltenen Objekten
- Strategie 1: sicherstellen, dass Objekte mit Guarded Suspension referenziert und von mehreren äußeren Objekten aus angesprochen werden können
- Strategie 2: bei Containment dürfen Methoden der inneren Klasse nur ohne Lock des äußeren Objektes aufgerufen werden. Bei Synchronisation der inneren Methode, äußere nicht synchronisiert -> Durchreiche-Muster und Hilfsmethoden mit jeweils neuen Threads

2. Konservative Policy: Balking

- Prüft bei Aufruf, ob Vorbedingung erfüllt ist. Falls nicht, gibt Methode Fehlermeldung zurück.
- Üblicher Mechanismus, sowohl in sequentiellen als in parallelen Programmen
- Inaction (keine Fehlermeldung, kein Feedback) ist einfacher Spezialfall.
- Implementierungsaspekte
  - betrachte Bedeutung und Konsequenzen der Ausnahme,
  - unterscheide Arten von Clients, die damit umgehen müssen
  - entwerfe Exception Types und globale Aufräum Aktionen
  - prüfe Vorbedingungen und Return bevor irreversible Aktionen innerhalb der Methode durchgeführt werden
  - prüfen der Vorbedingungen sollte möglichst ohne Synchronisation erfolgen
  - soweit möglich, schaffe Zugriffsfunktionen (public), so dass Clients Zustand vorab untersuchen/testen können

Timeout Designs

- Sind Mischungen aus Balking und Guarded Suspension
- In verteilten Applikationen mit unsicheren Kommunikationspartnern üblich
- Innerhalb der Programmierung mit Threads liefern sie eine zusätzliche Möglichkeit zum Debugging und zur automatischen Überwachung des Programms zu Laufzeit:
  - Timeouts liefern Verdachtsmomente für Deadlocks und andere Lebendigkeitsprobleme
- Grundidee: wait(long millisec) liefert nach k Zeiteinheiten einen Interrupt, sodass Client nicht beliebig lange auf Feedback warten muss
- Auswahl geeigneter Zeitkonstanten applikations- und plattformabhängig
- Implementierungshinweis:
  - auch bei Auftreten des Timeouts lohnt es sich, jeweils zuerst die Bedingung zu überprüfen, da zwischen Timeout und Bearbeitung des Threads weitere Threads durch die CPU bearbeitet werden können, so dass die Bedingung ggfs inzwischen eingetreten ist.

Zwischenübersicht

- Zustandsabhängige Aktivitäten
- Zustandsbeschreibung: logische Zustände vs reale Zustände,
- Reaktionsmuster
  - Pessimistisch, konservativ
    - Inaction: simple
    - Guarded Suspension: mit wait-notifyAll realisiert
    - Balking
  - Optimistisch
- Nachverfolgen von Zuständen
- Latches
- Nested Monitor Problem

Optimistische Kontrolle: Try&See Design

- Vorgehensweise:
  - nicht alle Vorbedingungen werden geprüft
  - Failure kann festgestellt werden
  - Undo Aktionen zum Rücksetzen auf den alten Status sind verfügbar
- wird in unterschiedlichen Bereichen eingesetzt:
  - Schemata zur Aktualisierung von Tabellen in Datenbanken, Fehlverhalten wg Interferenz relativ selten
  - Fehlertolerante Verfahren: zum Vergleich redundanter Berechnungen
  - Testability Designs, in denen Objekte Berechnungen zur Selbstüberprüfung vornehmen

Radikaler Neuansatz

1. Auf Synchronisation komplett verzichten
2. Interferenz kontrollieren, Failure detektieren
3. ggfs mit Undo Operationen alten Zustand wiederherstellen

Funktionalität: 3 Grundfunktionen

- Failure Detection, z.B.
  - durch Kontrolle logischer Zustände,
  - Auffangen von Exceptions
- Verhaltensstrategie für den Fehlerfall:
  - Inaction, Exceptions oder Retry
- Handlungstrategie für die Folgen des Fehlerfalles
  - Provisional Action: Durchführung der Aktion wird nur „vorgespiegelt“ und erst bei Erreichen eines positiven Ausgangs auch bestätigt.

    Implementierung z.B. mit Kopien der Daten auf „Shadow Variables“ und abschließendem Umsetzen von Referenzen oder Kopieren der Resultate.
  - Rollback/Recovery: die Durchführung von Aktionen muß rückgängig gemacht werden, d.h. vorherige Dateninhalte müssen wiederhergestellt werden, Nachrichten müssen durch „Antimessages“ in ihrer Wirkung aufgehoben werden.
- Für Aktionen, deren Auswirkungen irreversibel sind, scheiden optimistische Strategien aus !!!

Optimistische Kontrolle ohne Synchronisation

- Grundregeln
  - isoliere Zustandsänderungen zu atomaren Operationen

    d.h. im wesentlichen Zuweisungen von Referenzen oder Integer Werten
  - verfolge die aktuelle Version des betrachteten Zustands
  - breche Methoden ab, die nicht vollständig bearbeitet werden können oder die nicht zurückgesetzt werden können ohne Versionskonflikte zu erzeugen
  - arrangiere Wiederholungen oder erlaube Wiederholungen von Methoden die bisher nicht erfolgreich waren
- Grundidee optimistischer Kontrolle
  - isoliere alle Updates von Instanzvariablen zu einer einzelnen atomaren Update Methode, die jeweils als letztes Statement einer Methode aufgerufen wird

Optimistische Kontrolle ohne Synchronisation

- Grundidee optimistischer Kontrolle
  - isoliere alle Updates von Instanzvariablen zu einer einzelnen atomaren Update Methode, die jeweils als letztes Statement einer Methode aufgerufen wird
- Implementierung als commit / compareAndSwap Methode
  - Argumente: der angenommene und der neue Zustand des Objektes
  - Update erfolgt, falls angenommener Zustand und aktueller Zustand gleich (konsistent) sind
  - anderenfalls wird durch Mismatch Failure erkannt

```
Class Optimistic {
private State currentState_ ;

synchronized boolean commit (State assumed, State next) {
boolean success = ( currentState_ == assumed ) ;
if (success) currentState_ = next ;
return success ; }}
```

Optimistische Kontrolle ohne Synchronisation

- Commit bewirkt Versionen von stabilen Zuständen
- Wechsel erfolgen nur von einem stabilen Zustand zum nächsten.
- Wechselversuche können bei Mißerfolg unterbunden, abgebrochen oder wiederholt werden.

- Passendes Klassendesign erfordert 3 prinzipielle Schritte
  - Strukturierung des Zustands
  - Strukturierung der Methoden
  - Umsetzung einer Update Strategie (Ausnahmen/Exceptions, Wiederholungen, Synchronisation)

Strukturierung des Zustands

- Alle Instanzvariable des lokalen Zustands werden in einer Einheit als „Version“ des Zustands isoliert.
- Extremfall des Regroupings
- unterschiedliche Realisierungsmöglichkeiten
  - definiere ein unveränderliches Hilfsobjekt mit allen Werten der Instanzvariable. Die Host Klasse hält lediglich eine Referenz auf ein Hilfsobjekt, um den aktuellen Zustand zu verwalten

    Dies erschwert Spezialisierung, weil jeweils 2 Klassen bearbeitet werden müssen.
  - Definiere eine veränderbare Repräsentantenklasse, die zusätzlich eine Versionnummer (oder Transaktionsid oder Zeitstempel) verwaltet. Die Versionsnummer wird bei jedem Update inkrementiert. Die Versionsnummer ist der „angenommene“ Zustand für commit.
  - Bette alle Instanzvariable + Versionsnummer in die Host Klasse eine, definiere commit mit allen angenommenen und neuen Werten dieser Variable.
  - Und Kombinationen dieser Ansätze ...

Strukturierung von Updates

- Alle Zugriffsmethoden erfordern KEIN synchronized, weil eine aufgerufene Methode jeweils einen stabilen Zustand sieht.
- Die Update Methoden müssen folgende Struktur haben

```
State assumed = currentState() ;
State next = ...                    // Berechnung des jeweiligen Folgezustands
commit(assumed,next) ;    // ggfs Failure behandeln
... // weitere Aktionen, die vom neuen Zustand abhängen ohne ihn zu ändern
```

- Update Methoden dürfen vor commit keine irreversiblen Änderungen bewirken
- Update Methoden können bei Exceptions vor einem Commit ohne Rollback abbrechen
- commit kann dann kompliziert werden, wenn selbst weitere update Methoden aufgerufen werden (self-calls) oder wenn unabhängige Hilfsobjekte benutzt werden müssen

Behandlung von Exceptions

- Schema

```
State assumed = currentState() ;
State next = ...                    // Berechnung des jeweiligen Folgezustands
if (!commit(assumed,next))
  { rollback() ;                    // ggfs Failure behandeln
    throw new InterferenceException() ;
  }
... // weitere Aktionen, die vom neuen Zustand abhängen ohne ihn zu ändern
```

- Üblich: Commit Methode integriert bereits das Rollback
- klare, saubere Fehlerbehandlung: der resultierende Zustand ist derselbe, als wäre die Methode gar nicht aufgerufen worden
- Verantwortung bzgl Wiederholung etc liegt beim Client.

Behandlung von Exceptions mit Wiederholung

- Schema

```
for (;;) {
State assumed = currentState() ;
State next = ...
if (!commit(assumed,next))
   rollback() ;
else
  break ;
}
... // weitere Aktionen, die vom neuen Zustand abhängen ohne ihn zu ändern
```

- ohne Begrenzung der Versuche: Risiko eines Livelocks !!!
- Wait-free/lock-free Algorithmen: Klasse von Algorithmen für die Erfolg mit endlicher Anzahl Versuche bewiesen werden kann (leider wenige bekannt)
- Verkappte Variante des Busy Waiting
- Retry-Schleifen sollten yield() beinhalten
- bei direktem Zugriff auf lokale Variable müssen diese volatile sein!

Optimische Kontrolle ohne Synchronisation

- Achtung: commit Methode muß synchronized sein
- Synchronisation kann auch weiterhin verwendet werden (Mischformen)
- Insgesamt:
  - Ansatz erhält Lebendigkeit
  - Ansatz erhält Sicherheit
  - Ansatz erhält Potential für Parallele Berechnungen
  - Ansatz riskiert unnötige, erfolglose Berechnungen (Vergeudung von Rechenzeit)