Softwareentwürfe zur Kontrolle von Nebenläufigkeit:

Concurrency Control

Nach D. Lea, Kap. 5

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

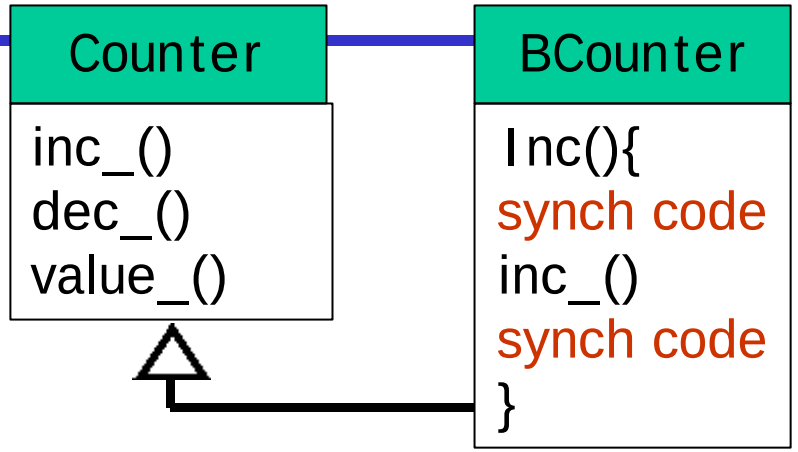
Heute:
Softwareentwürfe zur Kontrolle der Nebenläufigkeit

- Grundidee: Trennung von Arbeit (Operationen, Grundfunktionalität) und Kontrolle (Verwaltung, Zugriffskontrolle bei Threading)
- Grundstruktur: Schichtenmodell
- 3 Varianten
  - Subclassing: Hinzufügen von Kontrollstrukturen durch Spezialisierung
    - Zielt insbesondere auf Wiederverwendbarkeit und Modifizierbarkeit von Code ab
  - Adapter: Kontrolle delegierter Aktionen
    - Zielt auf Kapselung ab, Delegation, Wrapper
  - Acceptor: Repräsentation von Methodenaufrufen als Nachrichten
    - Zielt auf Kommunikation mit externen Prozessen ab
- Voraussetzung: Basisklassen müssen Funktionalität für Kontrolleingriffe bieten, z.B. Methoden zur Abfrage des aktuellen Zustands

Variante 1: Hinzufügen von Kontrolle durch Spezialisierung

- Pattern: Template Methode
  - Basisklassen liefern detaillierten Code in non-public Methoden oder default Code falls Methoden überladen werden sollen.
  - Public Methoden implementieren Synchronisationsstrategie, erbringen Funktionalität mittels non-public Basismethoden (Durchreichen)
- Public Methoden übermitteln also Nachrichten an Basismethoden und umgeben diese mit Vorher/Nachher Kontrollmaßnahmen
- 2 Varianten
  - Definiere Basismethoden in Basisklassen, ergänze Public Methoden in abgeleiteten Klassen.
  - Public Methoden in Basisklassen nutzten abstrakte Methoden oder Methoden mit Default Code, abgeleitete Klassen erbringen dann die jeweilige Funktionalität.

Beispiel: Counter und Bounded Counter



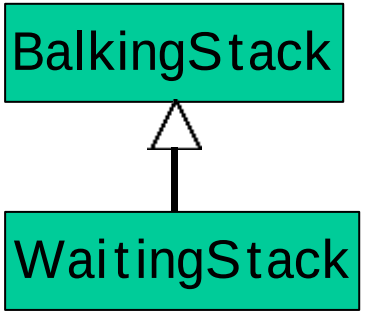

```
Public class Counter { protected long count_ ;
protected Counter(long c) { count_ = c ; }
protected long value_() { return count_  ; }
protected void inc_() { ++count_  ; }
protected void dec_() { --count_  ; }
}
public class Bcounter extends Counter implements BoundedCounter {
public Bcounter() { super(MIN) ; }
public synchronized long value() { return value_() ; }
public synchronized void inc() {
  while (value_() >= MAX) try { wait(); } catch (InterruptedException ex) {};
  inc_() ;
  notifyAll() ; }
public synchronized void dec() {
  while (value_() <= MIN) try { wait(); } catch (InterruptedException ex) {};
  dec_() ;
  notifyAll() ; }
}
```

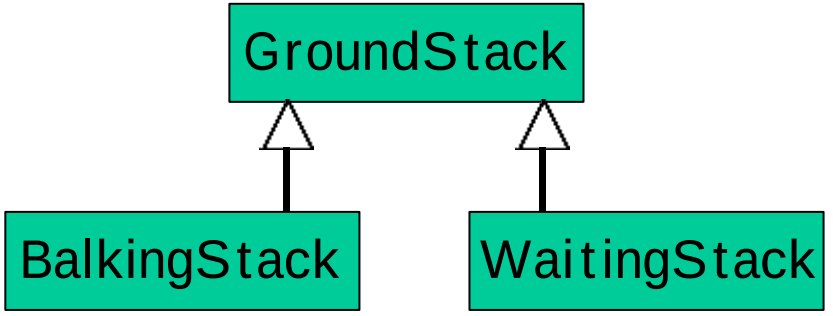
Anomalien bei Vererbung

- Vielfältige Möglichkeiten Produktivitätsgewinn durch Vererbung zunichte zu machen, z.B.
  - Falls die Basisklasse den Zustand nicht ausreichend repräsentiert und für eine abgeleitete Klasse zugreifbar macht, ...
  - Falls eine abgeleitete Klasse auf den Eintritt von Bedingungen wartet, für die die Basisklasse keine Nachrichten bereitstellt, ...
  - Falls die Basisklasse mit notify auskam, die abgeleitete Klasse jedoch notifyAll erfordert, ...
  - Falls eine Instanzvariable in der Basisklasse unveränderbar, in der abgeleiteten Klasse jedoch modifiziert werden soll, ...
  - ... müssen jeweils betroffene Methoden überarbeitet werden.
- Grundproblem: Hellseherische Fähigkeiten über zukünftige Nutzung einer Klasse nötig
- Abhilfe: Grundregeln in OO Design (Operationalisierung gelegentlich schwierig!)
  - Voreilige Optimierung sollte vermieden werden.
  - Design Entscheidungen sollten gekapselt werden.

Trennung von Funktionalität und Kontrolle in Schichten
Überwachung von Restriktionen

- Beispieldatenstruktur: Stack mit Methoden
  - boolean isEmpty(), void push(Object x), Object pop()
- Implementierung mit Balking wirft bei pop() Exception falls isEmpty() gilt
- bei Ableitung einer Klasse mit Warten (Guarded Suspension) im Falle von pop() bei leerem Stack
  - pop() wirft immer noch Exception, ohne es faktisch zu tun
- Besser:
  - ungeschützte Basisklasse mit Grundfunktionalität ohne Balking, etc
  - abgeleitete Klassen leisten Überwachung von Restriktion, Realisieren damit Policy wie Balking, Guarded Suspension, ...





Überwachung von nebenläufigen Zugriffen
Conflict Sets - Konfliktrelation

- Eine Konfliktrelation R über Aktionsmengen A, $R \subseteq A \times A$ beschreibt, welche Aktionen nicht nebenläufig ausgeführt werden dürfen.
- Beispiel: R = { (lesen, schreiben), (schreiben, schreiben) }
- nützlich, wenn Konflikte statischer Natur sind und zur Zeitpunkt des Designs an Paaren von Aktionen klar festgemacht werden können.
- Lösungsschema bei Kontrollfunktionalität in abgeleiteten Klassen:
  - deklariere Zählvariablen, um relevante, in Durchführung befindliche Aktionen aus A zu erfassen
  - umgebe Aufrufe von Methoden der Basisklasse mit Inkrement/Dekrement Operationen dieser Zähler
  - überwache Vorbedingung für den Aufruf von Methoden /Aktionen a der Basisklasse: alle Aktionen a´mit $(a,a´) \in R$ müssen terminiert sein
- bei optimistischen Kontrollverfahren: Konfliktrelation -> Invalidationsrelation, d.h. statt Vorbedingung wird Nachbedingung geprüft und Aktion bei Konflikt vor dem Commit verworfen.

Beispiel: R = {(a,b), (b,b)}, RawAB enthalte Methoden a_,b_

```
public class AB extends RawAB {
protected int anzA = 0 ;   protected anzB = 0 ;
public void a(Object p) {
  synchronized(this) {
     while (anzB > 0) try { wait() ; } catch (InterruptedException ex) {}
     anzA++ ; }
  a_(p) ;
  synchronized(this) { if (--anzA == 0) notifyAll() ; }
}
public void b(Object p) {
  synchronized(this) {
     while (anzB > 0 || anzA > 0) try { wait() ; } catch (InterruptedException ex) {}
     anzB++ ; }
  b_(p) ;
  synchronized(this) { if (--anzB == 0) notifyAll() ; }
}
```

Leser/Schreiber Pattern mit abgeleiteter Klasse

- „Leser“: Threads, die nur informationsbeschaffende Zugriffe durchführen
- „Schreiber“: Threads, die informations- und zustandsverändernde Zugriffe vornehmen, zusätliches Verhalten als Leser möglich
- Voraussetzungen
  - Basismethoden lassen sich in Lese- und Schreibmethoden aufteilen, wobei Schreibmethoden zusätzlich auch lesend zugreifen dürfen. Von derartigen Methoden dürfen unterschiedliche Ausprägungen auftreten.
  - Unterscheidung wird dadurch sinnvoll, dass mehrere simultane Lesezugriffe zulässig sein sollen, aber Schreibzugriffe nur im wechselseitigen Ausschluß.
- Technik:
  - Ableitung von Klassen für die Kontrolle der Nebenläufigkeit (Subclassing)
  - Kapselung von Zugriffen mit Vorher/Nachher Kontrollstrukturen
  - Zähler für aktive (oder wartende) Methodenaufrufe und Nachrichten

Leser/Schreiber Pattern mit abgeleiteter Klasse

- Designentscheidungen
  - Falls Leser aktiv sind, kann ein neuer Leser unmittelbar hinzutreten ?
    - Tradeoff: Starvation vs reduzierten Durchsatz, häufig: wenn Schreiber warten, werde neue Leser blockiert.
  - Falls Leser und Schreiber warten, wer darf zuerst ?
    - Möglich: statische oder dynamische Prioritäten, Zufallsentscheidung, häufig: Entscheidung Java Scheduler überlassen
  - Können aktive Leser sich ohne Freigabe von Locks dynamisch in Schreiber wandeln ?
    - Häufig: NEIN
- Implementierungstechnik
  - Definition eines Zustands und Nachverfolgen der Veränderungen,
    - Extremfall: eigener Scheduler
    - ausreichend: jeweils Anzahl wartender Leser und Schreiber erfassen.
  - In abgeleiteter Klasse Basismethoden mit Vorher/Nachher Kontrollstrukturen umgeben

Leser/Schreiber Pattern mit Basisfunktionalität in abgeleiteter Klasse

```
Public abstract class RW {
protected int activeR = 0 ;                ebenso activeW, waitR, waitW
protected abstract void read_() ;          ebenso wait_() ;
public void read()  { beforeR()  ; read_()  ; afterR()   ; }
public void write() { beforeW() ; write_() ; afterW() ; }

protected boolean allowR () { return waitW   == 0 && activeW == 0 ;}
protected boolean allowW() { return activeR == 0 && activeW == 0 ;}

protected synchronized void beforeR() { ++waitR ;
  while (!allowR()) try { wait() ;} catch (InterruptedException ex){}
  --waitR ; ++activeR ; }
protected synchronized afterR() { --activeR ; notifyAll() ;}

protected synchronized void beforeW() { ++waitW ;
  while (!allowW()) try { wait() ;} catch (InterruptedException ex){}
  --waitW ; ++activeW ; }
protected synchronized afterW() { --activeW; notifyAll() ;}
}
```

Adapter (auch Views, Wrapper)

- Alternative zu abgeleiteten Klassen
- sinnvoll, wenn
  - nicht alle Methoden aus einer/mehreren Basisklassen zugänglich sein sollen
  - Namen von Methoden, Signaturen angepaßt werden sollen
  - Identitätswechsel akzeptabel sind, da das Wrapper-Objekt eine andere Identität als das kontrollierte Objekt hat (z.B. bei Selbstaufrufen, return this Statements, ...)
  - die kontrollierte Klasse ausreichend Public Methoden, insbesondere für den Zugriff auf die Zustandsbeschreibung hat, da die Adapterklasse keine besonderen Rechte hat.
  - Kapselung vollständig ist, dh kein anderes Objekt als der Adapter kann auf das kontrollierte Objekte zugreifen. (Adapter weiß wenig über Interna des kontrollierten Objektes, daher sichert Containment Kontrolle der Nebenläufigkeit ab.)
- Nachteil: Annehmlichkeiten der Vererbung gehen verloren.

Adapter zur Synchronisation

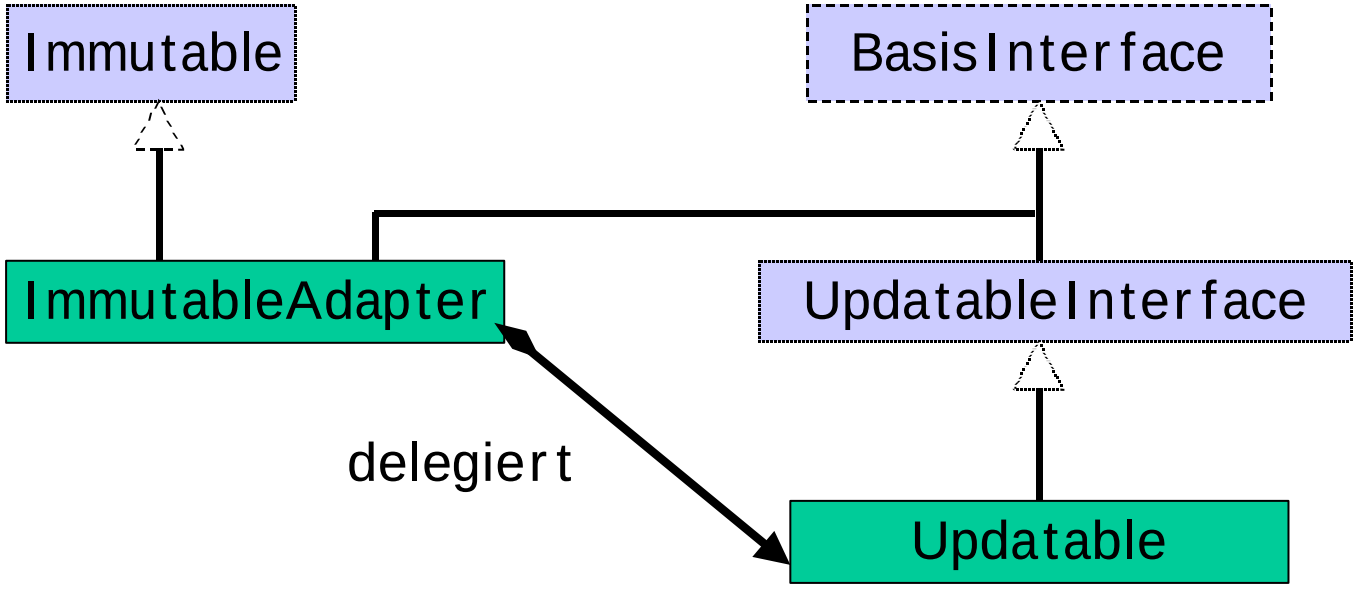
- Kapselung von unsynchronisierten Basisobjekten mit voller Synchronisation.
- Triviales Delegationsprinzip
- Mögliche Anwendung: „legacy“ Software, deren interne Funktionsweise unbekannt ist. Adapter schafft einzigen, sicheren Zugangspunkt.
- Implementierung:
  - private Instanzvariable mit Referenz auf das gekapselte Objekt. Zuweisung muß nicht fest sein, aber exklusiver Zugang muß garantiert sein.
  - Methodenaufrufe werden abgebildet, Parameter werden durchgereicht
    - Achtung Identität des gekapselten Objektes muß verborgen bleiben, Rückgabewerte „return this“ muß auf „return this“ abgebildet werden.
- Weil Adapter nicht abgeleitet ist, können Signaturen frei umgestaltet werden und auch Zugriffe auf Public Variable verhindert werden.
- Integration von Kontrollmechanismen wie z.B. Guarded Suspension innerhalb des Adapters durch Klammerung der durchgereichten Methoden mit Vorher/Nachher Methoden wie vorher auch.

Read-Only Adapter

- Gekapseltes Objekt wirkt dadurch wie ein unveränderliches Objekt.
- Adapter kontrolliert nicht Synchronisation sondern allein Sichtbarkeit.
- Sinnvoll, wenn
  - Kopieren großer Objekte als Alternative nicht sinnvoll ist
  - die Erstellung einer zusätzlichen Immutable Klasse für eine bestehende Klasse nicht sinnvoll ist
  - man sicherstellen kann, dass der Adapter den exklusiven Zugriff gewährleisten kann.
- Design Schritte
  - (Optionale Konvention) Definition eines ansonsten leeren Interfaces „Immutable“
  - Definition eines Basis Interfaces mit der Teilmenge der Funktionalität, die für das gekapselte Objekt keine Zustandsänderungen bewirkt.
  - Definition eines abgeleiteten Interfaces, das zustandsändernde Methoden ergänzt und Implementierung in problemspezifischer Weise.

Read-Only Adapter

- Weitere Design Schritte
  - Deklaration des unveränderlichen Adapters als final. Der Adapter implementiert Immutable und das Basisinterface.
  - Der Adapter erhält eine Instanzvariable des Basistyps, Methoden werden passend delegiert.



Adapter zur Erweiterung atomarer Aktionen

- Z.B. zur Integration von Kontrollausgaben an einer Schnittstelle
- Adapter
  - ist voll synchronisiert,
  - reicht Methodenaufrufe durch,
  - führt in Ergänzung jedes Methodenaufrufes weitere Aktionen durch, z.B. Trace Ausgaben, Ablaufs/Nutzungsprotokolle, Statistiken
- kann ohne Modifikation der gekapselten Klasse zwischengeschaltet werden
- Risiko: Nested Monitor Problem je nach interner Kontrollpolitik

Zwischenstand

- Bisherige Techniken
  - Subclassing
  - Adapter, Delegation
- Vorteile:
  - einfache Trennung von Funktionalität und Kontrolle der Nebenläufigkeit
- Grenzen, wenn mehr Abstraktion / Übersicht, Reflektion erforderlich ist:
  - Abhören / Verarbeiten von Nachrichten
  - Nachverfolgung des Zustands
  - Scheduling von Aktivitäten
- daher im weiteren: Reflektierende Techniken
  - Trennung von Basisfunktionalität und Kontrolle der Nebenläufigkeit
  - Grundidee: Übermittlung von Nachrichten (reifications) statt einfacher Methodenaufrufe
- typischer Anwendungsbereich: verteilte Systeme

Meta-Objekte: Akzeptoren, Event Frameworks

- Grundidee: Meta-Objekte nehmen Aufträge in Form von Nachrichten an, interpretieren und bearbeiten diese durch Basisklassen ähnlich wie bei Subclassing und Adaptern.
- Wesentliche Unterschiede
  - Kommunikation basiert auf Nachrichten, Ereignissen, Signalen
  - größere Komplexität, z.B. durch Scheduling oder durch Kaskadierende Ketten der Verantwortlichkeiten
- Beispiele für Nachrichtenarten
  - Integerwerte für Tastendruck, Mausklicks etc
  - Strings für eingehende Anforderungen und deren Argumente
  - eigene Event Class für Nutzereingaben wie in Java AWT
  - ein CORBA Request Packet von einem anderen Prozeß
  - ein ausführbares Objekt x mit der Aufgabe x.start aufzurufen
  - ein Klassenobjekt mit der Aufgabe eine neue Instanz zu erzeugen

Akzeptoren

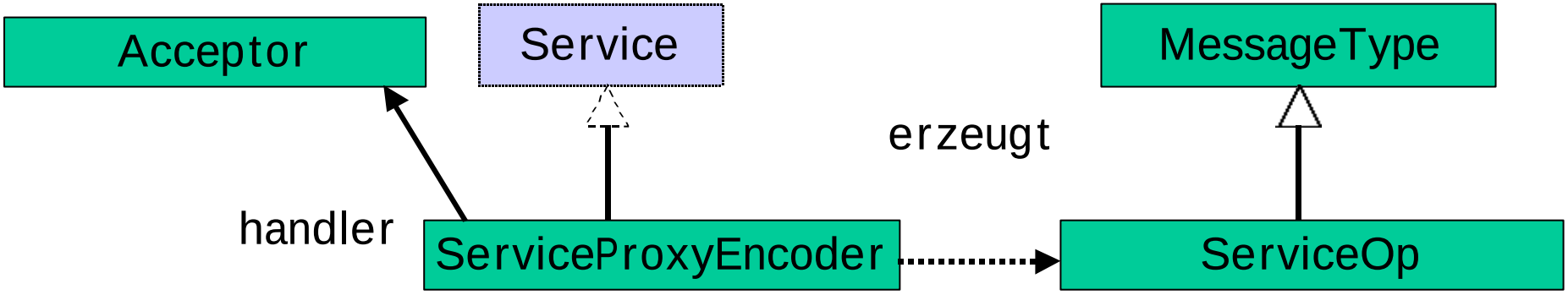
- Grundfunktionalität

```
Public interface Acceptor {
  public void accept( MessageType msg ) ; }
```

- Beispiele für Akzeptordesigns
  - Akzeptor unterstützt sowohl accept als auch Aufrufe für Basismethoden
  - Akzeptor delegiert accept an andere Objekte
  - Akzeptor hat eine Dispatch Tabelle zur Abbildung von Anforderungen zu Methodenaufrufen
  - Akzeptor hat ein Thread-per-message Design, je Anforderung eigener Thread.
  - Akzeptor repräsentiert und manipuliert Zustandsinformation explizit selbst oder in eigenen EventConditionAction Klassen
  - Akzeptor scheduled mehrere Agenten zur Ausführung von Request
  - Akzeptor führt History Logs für Undo und Redo Operationen.
  - Akzeptor filtert/transformiert Nachrichten bei Delegation an andere Akzeptoren

Proxy Encoder

- Eigene Klasse, kodiert Methodenaufruf durch passende Nachricht und ruft zugehörigen Akzeptor auf.

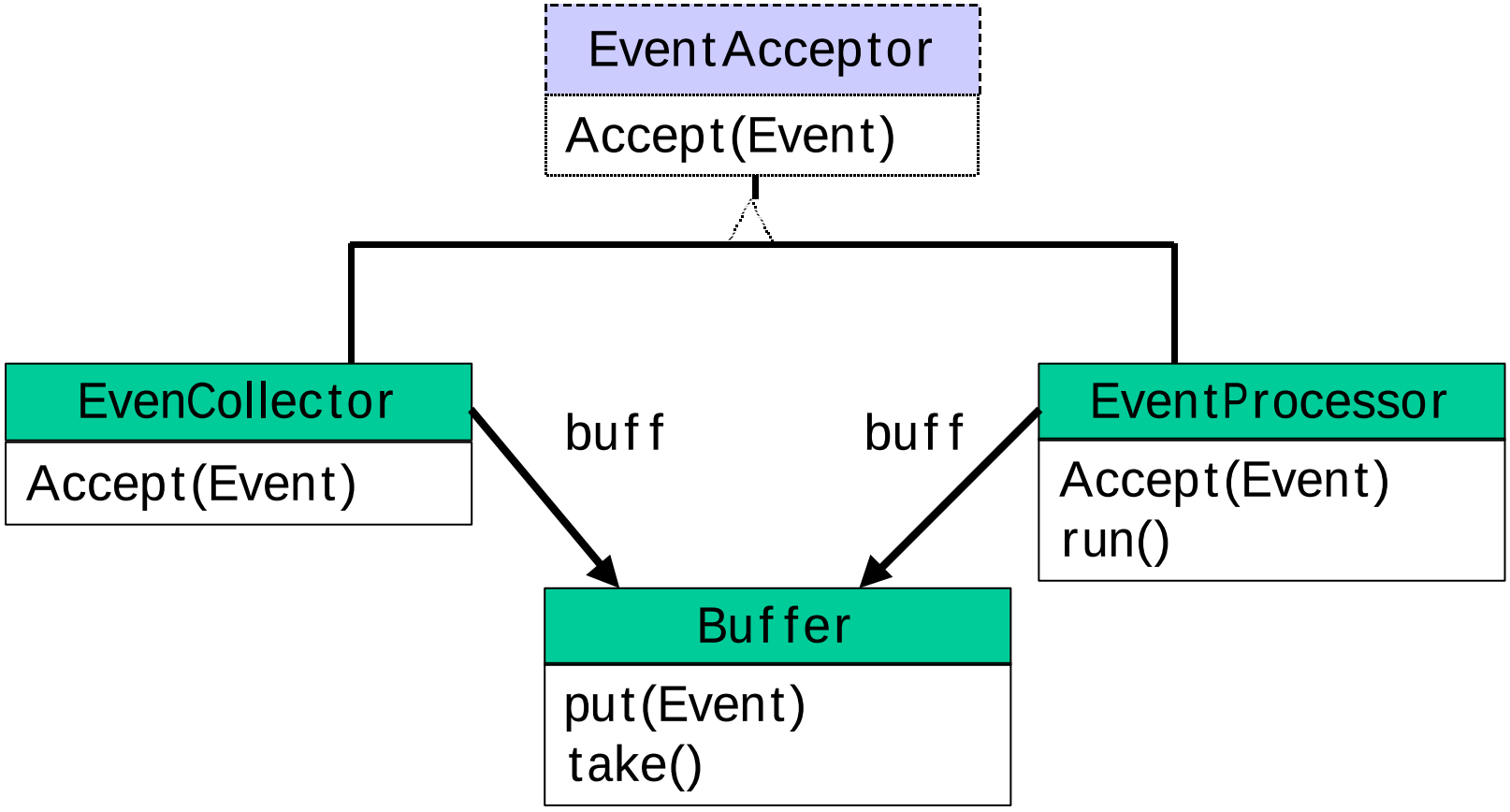



```
Public interface Service { public void op(Object arg);}
public class ServiceOp extends MessageType { private Object arg ; ...}
public class ServiceProxy implements Service {
  protected Acceptor handler_ ;
  public void op(Object arg) {
     MessageType m = new ServiceOp(arg) ;
     handler_.accept(m) ;
}}
```

Ereignisschleifen - Event loops

- Event Loops nehmen laufend Request Nachrichten an und leiten diese zur Verarbeitung weiter.
- Gepufferte Event Loops arbeiten asynchron zu den Auftraggeber Objekten, aber nicht notwendigerweise zu den Dienstleister Objekten.
- Implementierungsvarianten: a) je Request ein Thread, b) mit gepufferter Abarbeitung durch einen einzigen Thread
- Design
  - Event Klasse zur Charakterisierung der Nachrichten
  - Buffer Klasse zur Speicherung von Nachrichten
  - Event Collector Klasse, um eingehende Requests annehmen und puffern zu können.
  - Event Processor Klasse, um gepufferte Requests zu verarbeiten.

Ereignisschleifen - typische Klassenstruktur



Ereignisschleifen: Beispiel Code Fragmente

```
Public abstract class Event { public abstract int eventCode() ; }

public class EventCollector implements Acceptor {
  protected Bbuffer buff_ ;
  public EventCollector() { buff_=new Bbuffer(100);
     new EventProcessor(buff_) ; }
  public synchronized void accept(Event e) { buff_.put(e);}
}

public class EventProcessor implements Runnable, Acceptor {
  protected Bbuffer buff_ ;
  public EventProcessor(Bbuffer b) { buff_ = b ; new Thread(this).start();}
  public void accept(Event e) { switch (e.eventCode()) { ... }}
  public void run() { for (;;) accept((Event)(buff_.take())) ; }
}
```

Tabellenbasierte Eventhandler

- Accept Methode geht im einfachsten Fall nur eine statische Anzahl möglicher Ereignisse und zugehöriger Aktionen durch.
- Flexible Variante des EventProcessors
  - delegiert Aktionen an Eventhandler für einzelne Ereignistypen
  - verwaltet installierte Eventhandler mit den zugehörigen Ereignistypen in einer Tabelle, z.B. Hashtabelle mit Eintrag (Eventtype,Handler)
  - je Request wird für jeden passenden Eintrag der Tabelle der zugehörige Handler mit dem Event aufgerufen (unterschiedliche Varianten möglich)

Listeners

- Typisches Meta-Objekt in Verbindung mit externen Prozessen
- Listener besteht aus Interpreter Schleife, die permanent Nachrichten von einem Stream entnimmt und an passende ereignisverarbeitende Methoden weiterleitet.
- Variationen
  - leistet Aufgaben des Parsers und Builders durch Transformation von Eingaben in interne Repräsentation von Aufgaben
  - reicht den Eingabestrom ein den Execution Handler durch, damit dort Argumente und Daten extrahiert werden können
  - folgen einer thread-per-channel Politik, wobei für jeden neu geöffneten Kommunikationskanal ein eigener Thread erzeugt wird.
- Beispiel: listener für Webserver
- für einfache Anwendungsprogramme sind Meta-Objekte OVERKILL !

Zusammenfassung

- Concurrency Control
  - Subclassing
  - Delegation, Adapter
  - Meta-Objekte, Acceptors
- Welche Arten von Objekten treten bei Nebenläufigkeit auf?
  - Synchronised Objects: Methodenaufrufe werden lediglich um Synchronisation erweitert, um Interferenz zu vermeiden
  - Balking Objects: Objekte werfen ggfs Exceptions
  - Guarded Objects: Objekte können Ausführung von Methoden verzögern
  - Versioned Objects: Objekte können unterschiedliche Zustände vorgeben, liefern bei Methodenaufrufen nur bei Erfolg ein Commit
  - Concurrency Policy Controllers: Objekte repräsentieren und manipulieren explizit Zustände von Basisklassen.
  - Acceptors: Objekte, die Nachrichten darstellen, speichern und manipulieren können und diese auf Methodenaufrufe abbilden können.