Vorlesung
Software aus Komponenten

3. Komponentenmodelle

Prof. Dr. Hans-Gert Gräbe
Wintersemester 2004/05

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Sun und Java – der webzentrierte Ansatz

- Java: Geschichte und Konzepte
- Java und JavaBeans
- Modell
  - Eigenschaften
  - Ereignisse
  - Introspection
  - Anpassbarkeit
  - Auslieferung und Benutzung
- Weiterführende Entwicklungen
  - Enthaltungs- und Diensteprotokoll
  - Langzeit-Speicherung von JavaBeans
  - InfoBus
  - Bean MarkUp Language

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Java: Geschichte und Konzepte

große Erfolgsstory

- 1995/96 erste Anfänge (Sun Microsystems)
- heute (2003) eines der am häufigsten gebrauchten Schlagworte
- objektorientierte Programmiersprache, aber Magnet war Konzept von Applets, Mini-Applikationen und Funktionalität innerhalb von Webseiten

2 Ansätze, womit Java wirklich zur Killerapplikation wurde

- **Sicherheitskonzept**
  - Applet wird doppelt geprüft (Übersetzungszeit und Ladezeit)
  - strenge Sicherheitsregeln (security policies), die mit keiner anderen Programmiersprache erreicht werden
  - Sicherheit auch im compilierten Code eines JIT-Compilers
  - Sicherheitsaussagen durch Erzeugerzertifikate möglich
    - „signed applets“ (unter Nutzerkontrolle)

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- **Java virtual machine**
  - Plattformunabhängigkeit
  - großer Vorteil für Applikationen, die übers Web verteilt werden
  - Vorteil vor allem im Standard
    - Java class-File Format und Java JAR-Archiv-Format
- beides nicht neu, jedoch in der Kombination und zu diesem Zeitpunkt durchschlagend

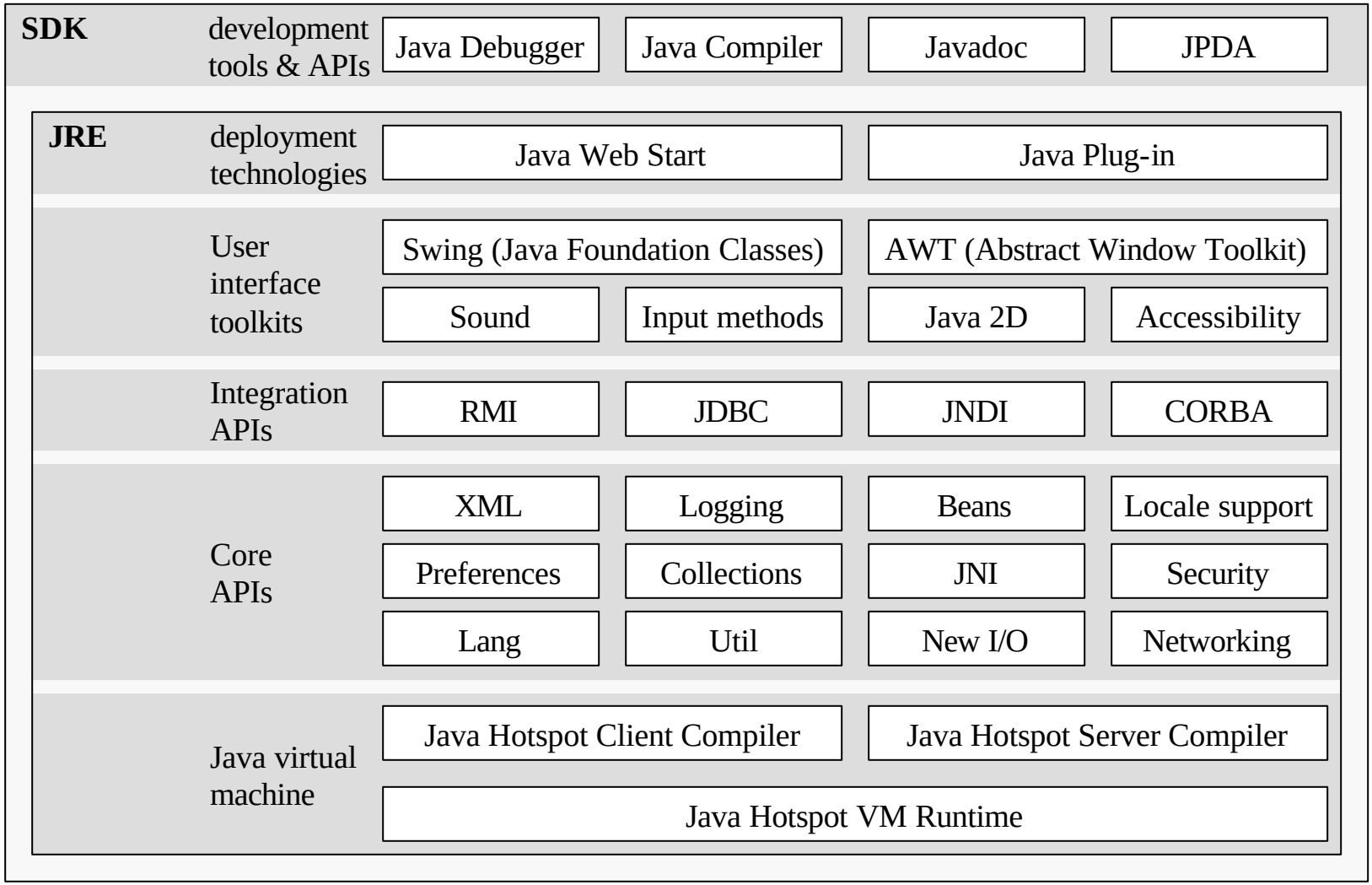
Java 2 (seit 1998)

- Fokus auf Applets aufgegeben
  - Applets heute nur noch marginal
- Plattform-Editionen
  - Funktionalitätsbündel für verschiedene Klassen von Nutzern
    - J2SE mit *JavaBeans* als Plattform für Einzelanwendungen
    - J2EE mit *Enterprise JavaBeans* (EJB) als Serverplattform (seit Ende 1999)
    - J2ME für mobile und eingebettete Anwendungen
- Formalisierung der Bezeichnungen Laufzeitumgebung (JRE), Entwicklungsumgebung (JDK) und Referenzimplementierung

Java 2 (Fortsetzung)

Referenzimplementierung der J2SE von Sun auf der Basis der HotSpot-JVM

- J2EE als Standard mit Implementierungen von verschiedenen (unabhängigen) Anbietern
  - (nicht laufzeitoptimierte) Referenzimplementierung von Sun als Beispiel-Implementierung im Quellcode verfügbar
- Prüfung der Unterstützung von Standards durch Kompatibilitätstest-Reihen
- Java BluePrints als Sammlung von Design-Richtlinien und Mustern, um spezielle Technologien zu unterstützen



Aufbau der Java 2 Plattform, Standard Edition v1.4 (Quelle: java.sun.com)

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

J2EE Architektur

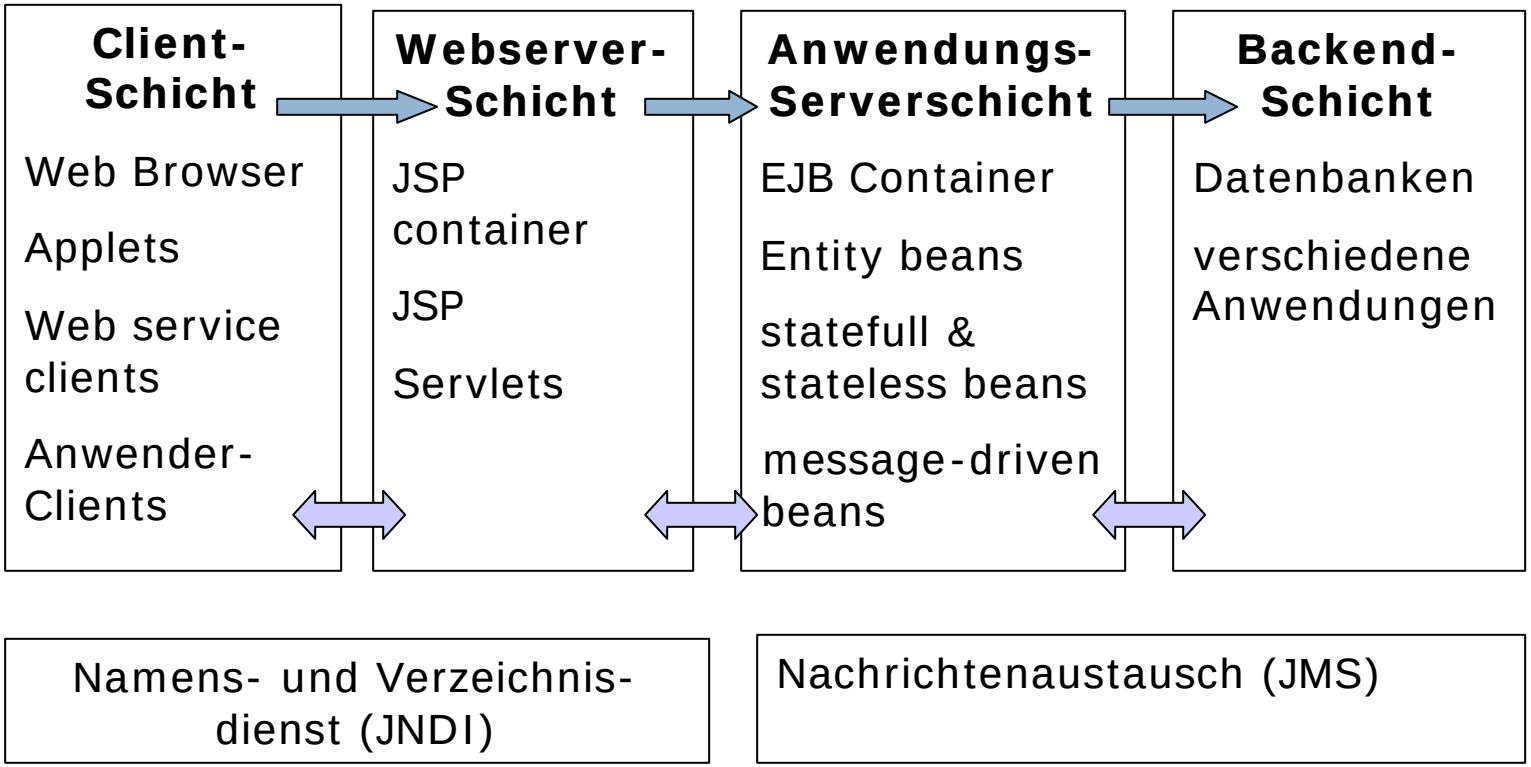
- Im Zentrum steht **Familie von Komponentenmodellen**
  - Client-Schicht: Anwenderkomponenten, JavaBeans, Applets
  - Webserver-Schicht: Servlets, JSP
  - Anwendungsserver-Schicht: EJB in vier Varianten (stateless session, statefull session, entity, message-driven session)

- JavaBeans kommt in allen Schichten zum Einsatz

- Integrations-Ebenen (Basisdienste):
  - Namens- und Verzeichnis-Infrastruktur (naming and directory interface, JNDI) sowie Nachrichten-Infrastruktur (Java messaging service, JMS) bilden die Klammern zwischen den verschiedenen Schichten
  - weitere I.-E.: Transaktionskoordinierung, Sicherheitsdienste

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

- J2EE Architektur
  - Kontrollfluss:
  - Datenfluss:



UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Wichtige von Java unterstützte Grundkonzepte

- **Methoden** (Verhalten, behavior) und **Attribute** (Status, state)
- **Schnittstellen:** Es können Interfaces und abstrakte Klassen definiert werden, die später (mittels ‚*implements*') implementiert werden sollen
  - Mehrfachvererbung von Schnittstellen (ohne Status und Verhalten)
- **Klassen:** Implementierungen von Schnittstellen. Es können von bestehenden Klassen spezielle Unterklassen (mittels ‚*extends*') abgeleitet werden.
  - Einfachvererbung von Implementierungen
  - vermeidet das Diamant-Problem
  - unveränderbare Implementierungen (final class, method, attribute)
- **Pakete** und **Pakethierarchien** als Modularisierungskonzept jenseits von Klassen
  - Namensgebung und Namensräume auf dieser Basis
  - keine Unterstützung von Mehrfachversionen
  - company-name.productname Präfix als Standard
  - Namensraum-Importe

- **Sichtbarkeitsklassen** von Attributen und Methoden (default, public, protected, private)
- **Ausnahmebehandlung** (exception handling): Es besteht die Möglichkeit, über Ausnahmen (mittels ‚try-catch'-Blöcken) vom Standard-Kontrollfluss abzuweichen
- **Threads und Synchronisation:** Nebenläufige Programmabläufe lassen sich mit Threads erzeugen und synchronisieren
- **Garbage Collection:** Nicht mehr referenzierte Objekte werden automatisch auf kontrollierbare Weise (‚*finalize*') zerstört
  - Anwender hat darauf aus Sicherheitserwägungen keinen Einfluss
- **Objektserialisierung:** Objekte, welche die Schnittstelle *Serializable* implementieren, können in einen Datenstrom geschrieben oder aus einem solchen aufgebaut werden (z.B. Speichern in eine Datei)
- **Ereignisse (events):** werden einige Folien später genauer besprochen

Grundlagen zu JavaBeans

- Komponentengedanke wird von Java nicht direkt unterstützt.
- JavaBeans ist die Spezifikation *eines* Komponentenmodells für Java
  - 1996 von Sun Microsystems eingeführt in Java Version 1.1
  - fasst mehrere Klassen und Ressourcen zusammen
  - keine klare Unterscheidung zwischen Beans und Beans-Instanzen
- Ziel: Das Zusammenfügen von Komponenten mit Hilfe von **visueller Programmierung** zu ermöglichen
  - Komponenten können einfach mit der Maus »zusammengeklickt« und in anderen Java-Applikationen oder in Java-Applets innerhalb von Web-Browsern eingesetzt werden
  - Design-Zeit und Laufzeit
- jede Java-Klasse ist im Prinzip eine JavaBean
  - um jedoch die **Kompositions- und Anpassungsfähigkeit** einer JavaBean optimal zu unterstützen, müssen gewisse Konventionen beachtet werden
  - JavaBeans sind damit plattformunabhängig, es muss lediglich eine Java Virtual Machine auf dem Zielsystem existieren

Charakterisierung von JavaBeans

- JavaBeans sind Mengen gewöhnlicher Java-Klassen, die zusätzlichen Standards folgen, um sie mit Werkzeugen, etwa in einem Beans-Builder, zu manipulieren:
  - ***Eigenschaften (properties)*** = Attribute, auf die über get- und set-Methoden nach einem festgelegten Muster zugegriffen werden kann.
  - ***Ereignisse (events)***, die von der Bean erzeugt werden und über einen Ereignisbeobachter (event listener) an andere Bean weiter gegeben werden können.
  - ***Introspektion***, ein Mechanismus, über welchen Methoden, Eigenschaften und registrierte Ereignisse der Bean abgefragt werden können.
  - ***Anpassung***, ein Mechanismus, mit dem Eigenschaften einer Bean mit einem Werkzeug konfiguriert werden können.
  - ***Persistenz***, ein Mechanismus, mit dem die Einstellungen einer Bean dauerhaft gespeichert und wieder geladen werden können.

Eigenschaften

- Konzeptionell: benanntes Attribut, dessen Wert das Aussehen oder Verhalten der Bean beeinflussen kann.
- Real: set- und get-Methode der Beans-Schnittstelle
  - interne Realisierung ist egal, da verborgen
- Wert kann gesetzt oder ausgelesen werden
  - durch Skripting oder Eigenschaftseditoren zur Assemblierzeit
  - durch set- oder get-Methoden zur Laufzeit
- typischerweise sind Eigenschaftswerte persistent
- Unterscheide
  - einfache Eigenschaften
  - indizierte Eigenschaften
  - gebundene Eigenschaften
  - Eigenschaften mit Nebenbedingungen
- Eine JavaBean kann mit Editoren für einzelne Eigenschaftstypen kommen
  - Editor muss Schnittstelle java.beans.PropertyEditor implementieren
  - java.beans.PropertyEditorManager übernimmt Zuordnung und Suche

- einfache Eigenschaften (simple properties)
  - realisiert als Attribut der Bean-Klasse durch Methoden
    - public <Eigenschaftstyp> get<Eigenschaftsname>( )
    - public void set<Eigenschaftsname> (<Eigenschaftstyp> a)
  - für Eigenschaften vom Typ *boolean* auch:
    - public boolean is<Eigenschaftsname>()
  - aus diesen Signaturen kann <Eigenschaftsname> und <Eigenschaftstyp> durch Introspektion ausgelesen werden

- indizierte Eigenschaften (indexed properties)
  - analog, aber konzeptionell Feld von Werten, die über einen Index angesprochen werden, deshalb Signatur
    - public <Eigenschaftstyp> get<Eigenschaftsname>(int index)
    - public void set<Eigenschaftsname>(int index, <Eigenschaftstyp> a)
  - unbekannt, ob intern wirklich als Feld implementiert

- gebundene Eigenschaften
  - Änderungen von Eigenschaften werden registrierten externen Beobachtern über PropertyChangeEvent mitgeteilt

```
void set<Eigenschaft>(.. newValue) {
    /* hole oldValue und setze newValue */
    /* feuere PropertyChangeEvent(this,eigenschaft,oldValue,newValue) */
}
```

  - Beobachter müssen sich dazu bei der Quelle registrieren, d.h. die Quelle muss Listener-Verwaltung implementieren
    - public void addPropertyChangeListener (PropertyChangeListener x)
    - public void removePropertyChangeListener (PropertyChangeListener x)
  - Beobachter müssen die Schnittstelle PropertyChangeListener implementieren, also die Methode propertyChange(e), welche das PropertyChangeEvent e auswertet.

- Eigenschaften mit Nebenbedingungen
  - wie gebundene Eigenschaft, aber Beobachter kann Veto einlegen und damit eine Änderung des Eigenschaftswertes zu unterbinden
    - Quelle und Beobachter nicht im gleichen Kontrollfluss, also nur über eine Ausnahme realisierbar: PropertyVetoException
  - Die set-Operation muss folgende Signatur besitzen:

    public void set<Eigenschaftsname> (<Eigenschaftstyp> wert)
    throws PropertyVetoException
  - Quelle muss Listener-Verwaltung implementieren

    public void addVetoableChangeListener (VetoableChangeListener x)

    public void removeVetoableChangeListener (VetoableChangeListener x)
  - Beobachter muss Schnittstelle VetoableChangeListener implementieren,also die Methode

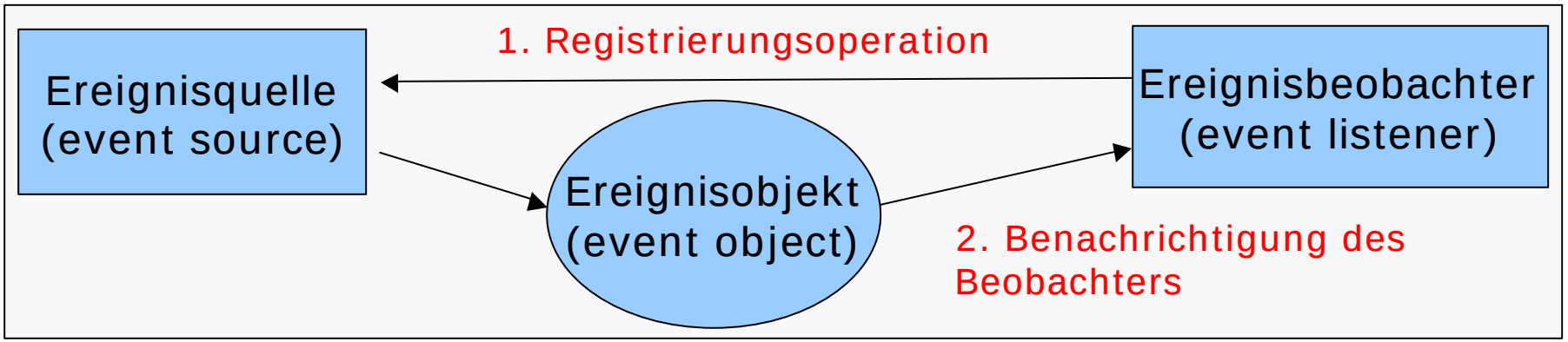
    void vetoableChange(e) throws PropertyVetoException

    die im Vetofall eine PropertyVetoException auslöst

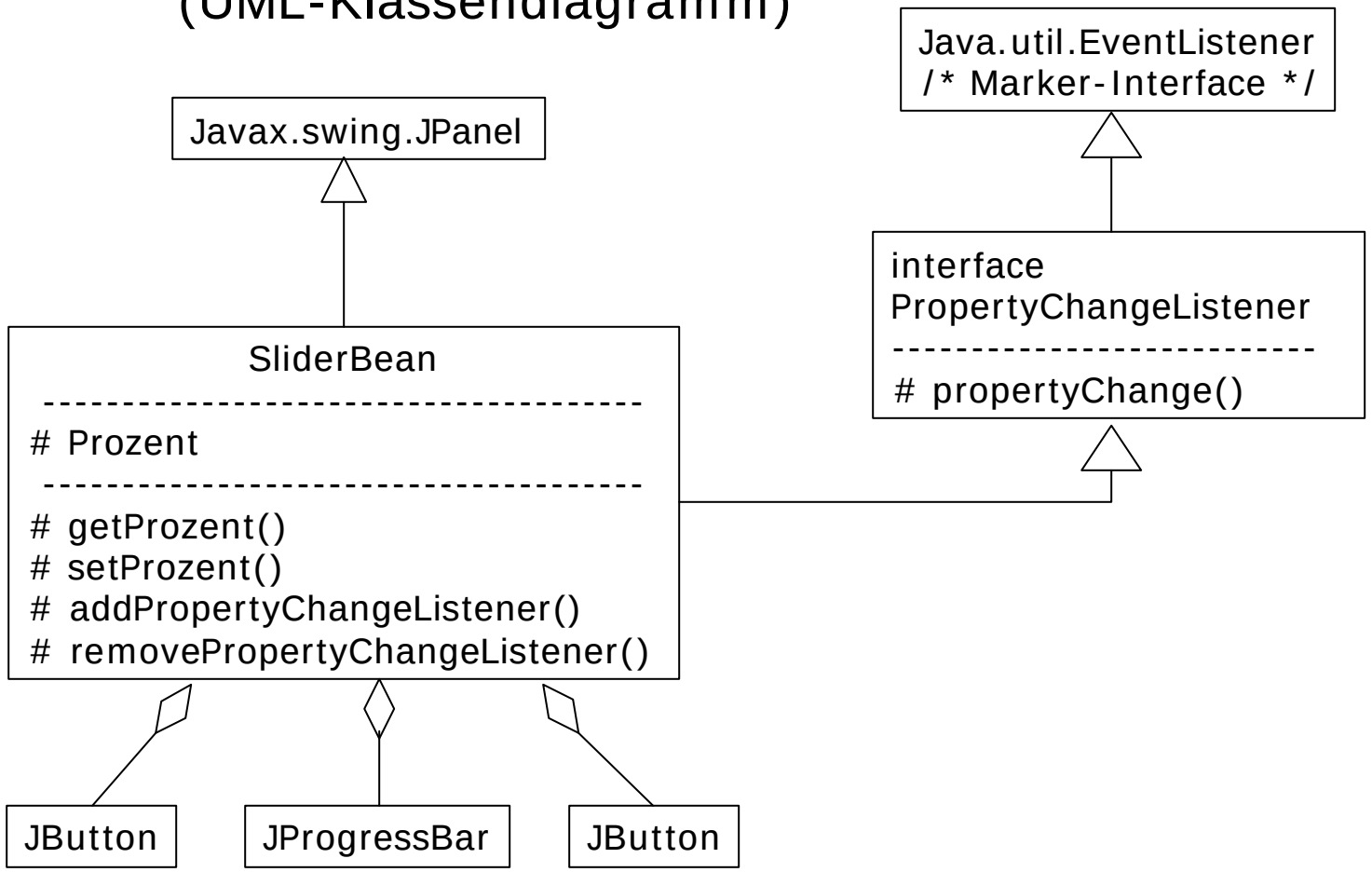
UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Ereignisbehandlung und JavaBeans

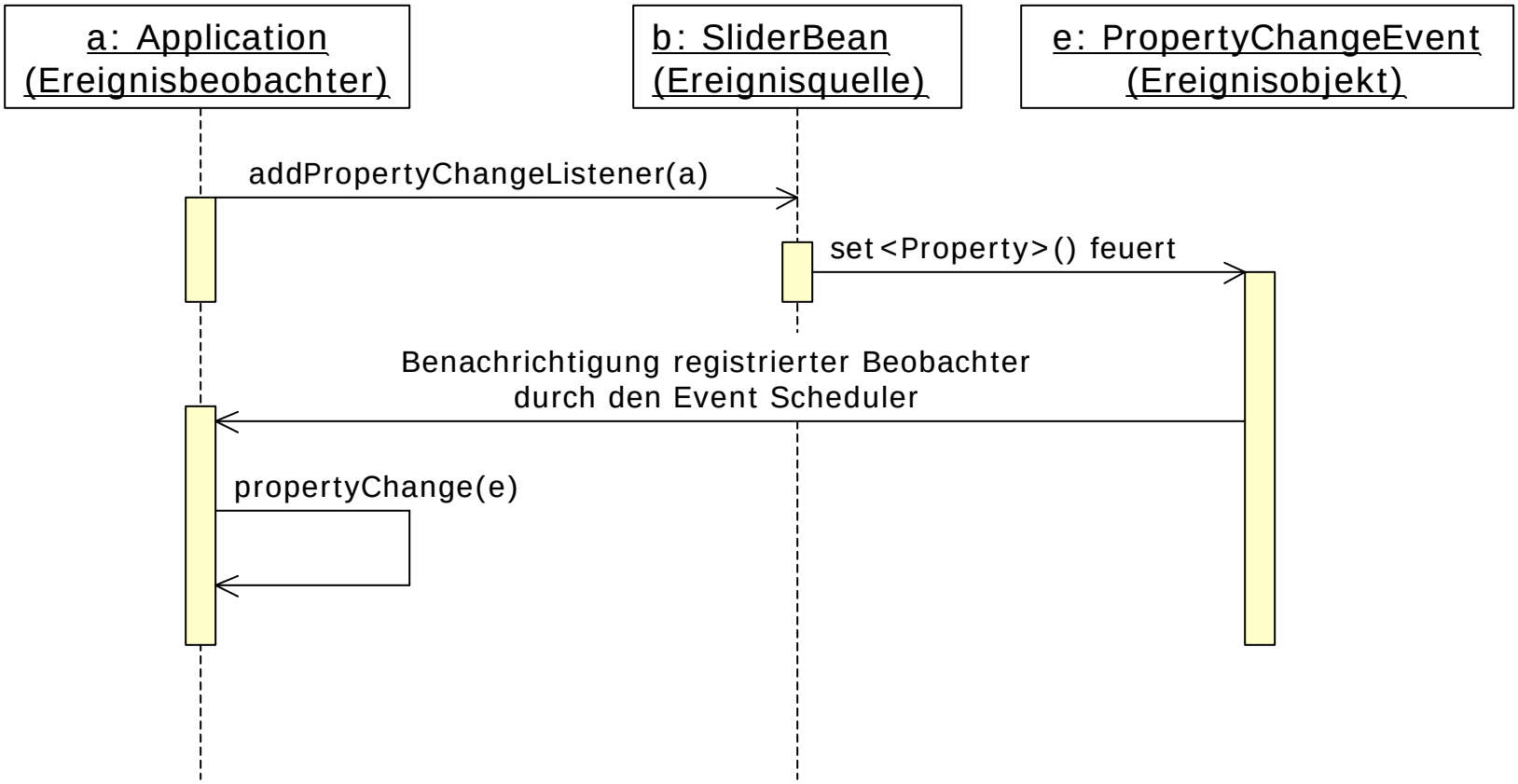
- JavaBeans benutzen das Ereignismodell von Java
- Ereignisse sind Objekte, die von einer Ereignisquelle erzeugt werden und an alle im Moment angemeldeten Ereignisbeobachter propagiert werden
- Ereignisobjekte sollten keine Änderung ihrer Attribute zulassen
- eine Komponente kann sich zur Laufzeit bei einer anderen Komponente als Beobachter an- und abmelden
  - public void add<Beobachtertyp>(<Beobachtertyp> l)
  - public void remove<Beobachtertyp>(<Beobachtertyp> l)
- Beobachter müssen das zugehörige Ereignis-Interface implementieren



Beispiel für das Ereignis-Design von JavaBeans (UML-Klassendiagramm)



(UML-Sequenzdiagramm)



Ereignis-Propagation

- es existieren zwei Ereignisquellenarten
  - mehrere Beobachter anmeldbar (multicast event source, Standard)
  - nur ein Beobachter anmeldbar (unicast event source)

- Multicast-Probleme:
  - Menge der Listener kann sich während der Ereignispropagation ändern
  - Ausnahme kann während der Ereignispropagation ausgelöst werden
  - Reihenfolge von Ereignissen: Listener kann eigene Events als Antwort auslösen.
    - Problem der Locks auf Ressourcen, die mit einem Event verbunden sein können, die den Zugriff durch Event-Verarbeiter regeln.
  - subtiles semantisches Problem: wie „materiell“ sind Events?

JavaBeans und Design-Pattern

- spezielle Namensgebung und Introspektion erlaubt unabhängige Werkzeugentwicklung (Designprinzip „Methoden-Pattern“)
  - Werkzeuge finden so heraus, welche Eigenschaften definiert sind
  - durch visuelle Programmierung zum Beispiel mit Ereignisbeobachtern verknüpft werden
- Java weicht damit von eigenen Sprachstandards ab
  - Bsp: java.util.EventListener als Marker-Interface
    - EventListener-Klassen erkennbar am ... implements EventListener
  - mögliche Java-konforme Lösung:
    - Marker-Interface Property
    - Konkrete Eigenschaften als Interface

```
interface <SpecialProperty> extends Property {
        public <PropertyType> get<SpecialProperty>();
        public void set<SpecialProperty>(<PropertyType> e);
}
```

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

JavaBeans entwerfen und nutzen

Spezifikation der Eigenschaften

- die Eigenschaften sollten möglichst beim ersten Entwurf genau spezifiziert werden, um Änderungen an der Schnittstelle zu vermeiden.
  - Problem der Änderung der Schnittstelle bei Versionswechsel
- vom Methoden-Pattern kann abgewichen werden, wenn die Schnittstelle java.beans.BeanInfo implementiert ist
  - dazu sind verschiedene Beschreibungen (Instanzen von PropertyDescriptor, EventSetDescriptor, MethodDescriptor, BeanDescriptor) in vorgegebenen Formaten anzufertigen

Beans-Lebenszyklus

- Für JavaBeans können die Phasen Entwurf, Auslieferung, Entpackung (von Beans), Anpassung an lokale Erfordernisse, Persistenzmanagement (von Beans-Instanzen) unterschieden werden
- Die Rollen des Beans-Entwicklers und des Beans-Nutzers lassen sich unterscheiden

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Anpassung an lokale Bedingungen

- JavaBeans müssen nach Übersetzung und Auslieferung an Bedürfnisse in gewissem Maße anpassbar sein

2 Möglichkeiten:

- Einsatz eines Werkzeugs zur Bean-Assemblierung
  - Eigenschaften-Editor des Entwicklungswerkzeuges benutzen
- die JavaBean stellt eigene Klasse (Implementierung der Schnittstelle Customizer) zur Verfügung, die eine Anpassung komplexer Eigenschaften ermöglicht

Auslieferung und Benutzung

- die JavaBean mit allen benötigten Ressourcen wird in einer Archiv-Datei mit der Endung .jar verpackt, um sie als Einheit verschicken zu können
- will man eine JavaBean in einem Entwicklungswerkzeug verwenden, so muss man in der Regel zunächst mit dem Entwicklungswerkzeug die .jar Datei der Komponente einlesen
- das Entwicklungswerkzeug merkt sich dann den Pfad, unter dem das Archiv zu finden ist und welche Komponenten in dem Archiv enthalten sind.

Weitere Konzepte

Containment and services protocol

- unterstützt logische Einbettung von JavaBeans-Instanzen
- Dadurch kann die JavaBean nicht nur Dienste der JavaVM und der Core APIs in Anspruch nehmen, sondern auch zusätzliche Dienste des Containers, in dem sie sich zur Laufzeit befindet
- genauso kann der Container die Dienste der eingebetteten JavaBean erweitern oder einschränken

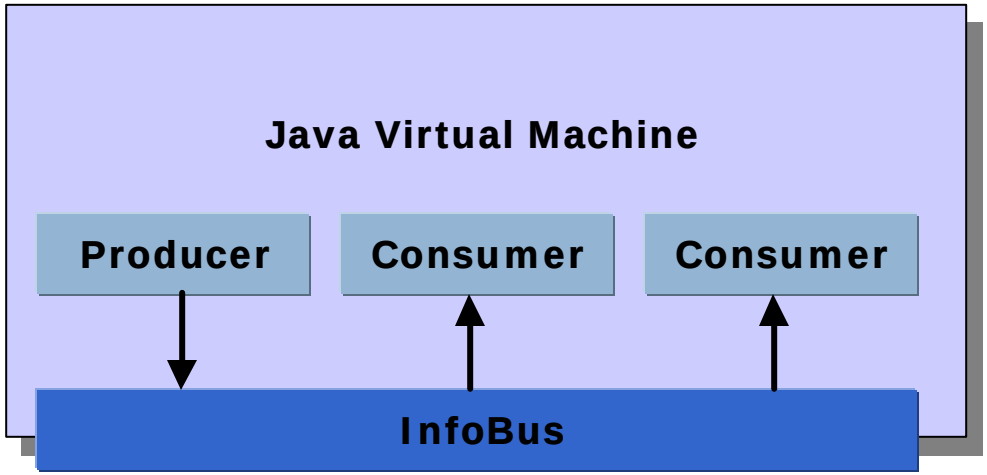
Archivierung einer JavaBean

- Alternative zum Serialisierungsprozess von Java
- Format ist XML mit DTD oder ein anderes proprietäres Dateiformat
- dabei wird nicht wie bei der Serialisierung alles gespeichert, sondern nur ein Teil der internen Objektdaten (public properties)
- so kann ein neueres Objekt trotz anderer Attribute mit den „alten“ Daten etwas anfangen, was bei Serialisierung nicht geht (da dort nur der komplette Zustand des Objekts wiederhergestellt werden kann)
- dadurch wird eine Art „Versionsunabhängigkeit“ erreicht

UNIVERSITÄT LEIPZIG
Institut für Informatik
Betriebliche Informationssysteme

Infobus

- gemeinsame Entwicklung von Sun und Lotus
- einfaches Verbinden von JavaBeans innerhalb einer VM über standardisierte Schnittstellen
- lose Kopplung zwischen den Komponenten
- ***Data-Consumer***: ruft Daten ab
- ***Data-Producer***: bietet Daten an und informiert über Änderungen
- ***Data-Controller***: wird zur Verarbeitung von Daten zwischen Producer und Consumer geschaltet



Bean Markup Language

- Entwicklung von IBM
- Scriptsprache um Beans zu erzeugen, zu konfigurieren und zu verbinden
- XML-basierte Syntax (wird durch XML-DTD beschrieben)
- 2 Ausführungsmodelle
  - in Java geschriebener Player, der BML-Skripte interpretativ ausführt
  - Compiler, der BML-Skripte in Java umsetzt (Geschwindigkeitserhöhung)

```
<bean class="java.awt.Frame"  id="topFrame ">
<property  name="title"  value="Testfenster"/>
<property  name="background"  value="0xeeeeeee"/>
<property  name="layout">
        <bean  class"java.awt.BorderLayout"/>
</property>
```