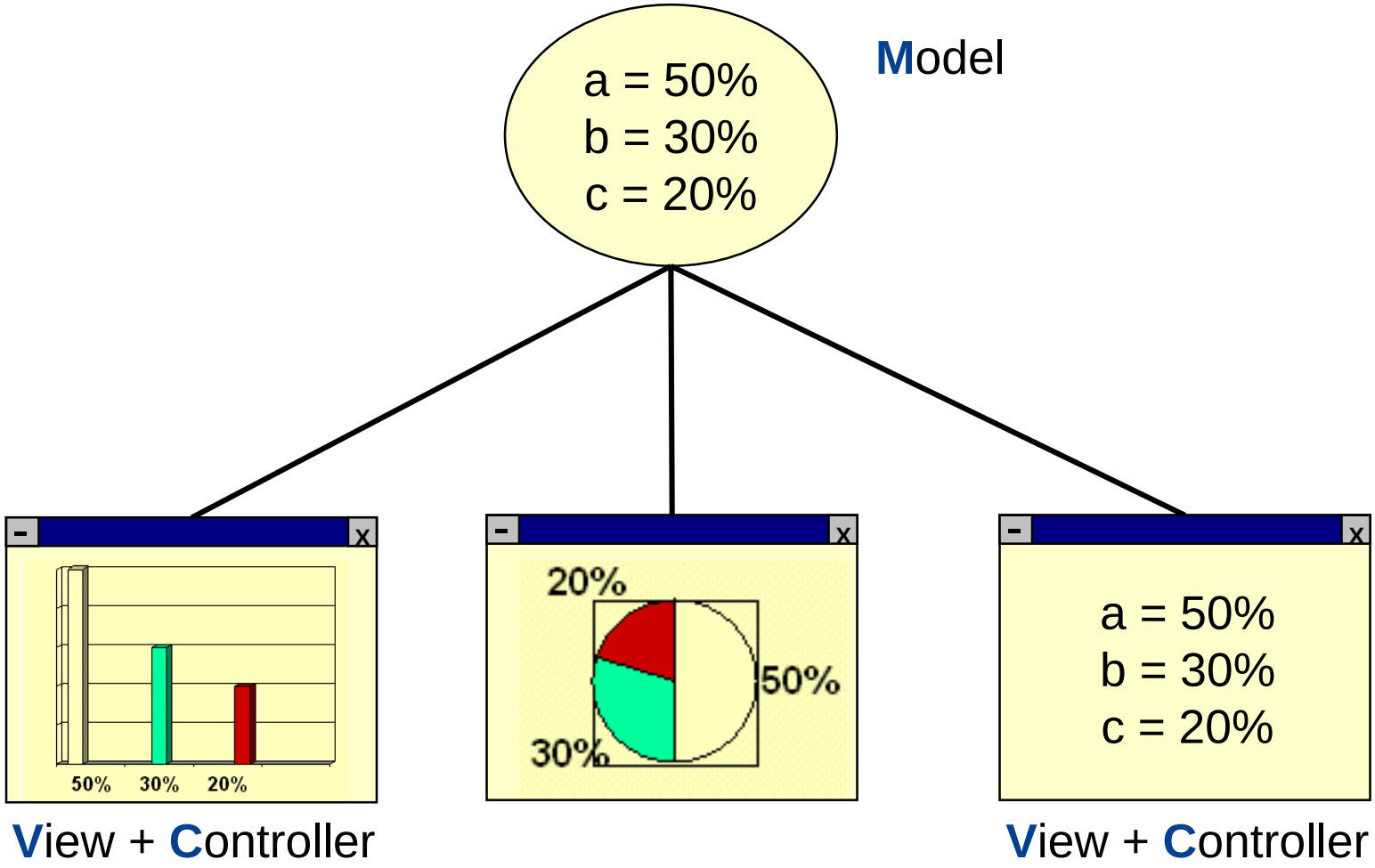
Kapitel 8

Entwurfsmuster (“Design Patterns”)

Stand: 26.1.2020

**Ergänzungen des Factory Method Patterns,
Visitor Pattern entfernt,
Bemerkung in Composite (F. 96) eingefügt**

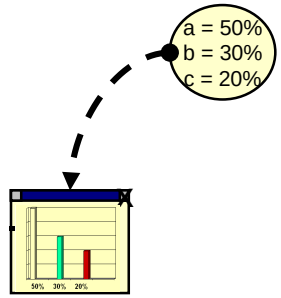
Beispiel:
Das MVC-Framework in Smalltalk



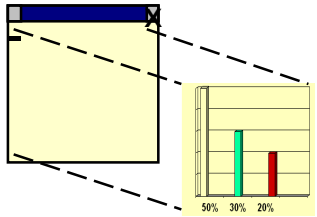
Neue Views können ohne Änderung des Modells oder der anderen Views hinzugefügt werden

Beispiel: Das MVC-Framework in Smalltalk

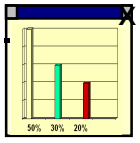
- Propagierung von Änderungen: Observer Pattern
  - Synchronisation bei Änderungen
  - Kommt z.B. auch bei Client/Server-Programmierung zur Benachrichtigung der Clients zum Einsatz

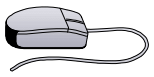


- Geschachtelte Views: Composite Pattern
  - View enthält weitere Views, wird aber wie ein einziger View behandelt.
  - Kommt z.B. auch bei Parsebäumen im Compilerbau zum Einsatz (Ausdrücke).



- Reaktion auf Events im Controller: Strategy Pattern
  - Controller können zur Laufzeit gewechselt werden.
  - Kommt z.B. auch bei der Codeerzeugung im Compilerbau zum Einsatz (Code für verschiedene CPUs).





Entwurfsmuster (Design Patterns)

Grundidee

- Dokumentation von bewährten Lösungen wiederkehrender Probleme.
- Sprache, um über Probleme und ihre Lösungen zu sprechen.
- Katalogisierungsschema um erfolgreiche Lösungen aufzuzeichnen.

Definitionsvorschlag

"Each pattern is a three-part rule that expresses a relation between

- a context,
- a system of forces that occur repeatedly in that context, and
- a software configuration that allows these forces to resolve themselves."

Richard Gabriel, on the Patterns Home Page

Pattern-Beschreibung

- Name(n) des Patterns
- Problem, das vom Pattern gelöst wird
- Anforderungen, die das Pattern beeinflussen
- Kontext, in dem das Pattern angewendet werden kann

- Lösung. Beschreibung, wie das gewünschte Ergebnis erzielt wird
  - Varianten der Lösung
  - Beispiele der Anwendung der Lösung

- Konsequenzen aus der Anwendung des Patterns

- Bezug zu anderen Patterns
- Bekannte Verwendungen des Patterns

Bestandteile eines Patterns: Kontext, Problem, Randbedingungen

- Problem
  - Beschreibung des Problems oder der Absicht des Patterns
- Anwendbarkeit (Applicability) / Kontext (Context)
  - Die Vorbedingungen unter denen das Pattern benötigt wird.
- Anforderungen (Forces)
  - Die relevanten Anforderungen und Einschränkungen, die berücksichtigt werden müssen.
  - Wie diese miteinander interagieren und im Konflikt stehen.
  - Die daraus entstehenden Kosten.
  - Typischerweise durch ein motivierendes Szenario illustriert.

Bestandteile eines Patterns: Lösung

Jede Lösungsvariante

Die Lösung (Software Configuration) wird beschrieben durch:

- Rollen
  - Funktionen die Programmelemente im Rahmen des Patterns erfüllen.
  - Interfaces, Klassen, Methoden, Felder / Assoziationen
  - Methodenaufrufe und Feldzugriffe
  - Sie werden bei der Implementierung aufg konkrete Programmelemente abgebildet (‚Player')
- Statische + dynamische Beziehungen
  - Klassendiagramm, dynamische Diagramme, Text
  - Meistens ist das Verständnis des dynamischen Verhaltens entscheidend
    - ⇨ Denken in Objekten (Instanzen) statt Klassen (Typen)!
- Teilnehmer – Participants
  - Rollen auf Typebene (Klassen und Interfaces)
- Beispiele der Anwendung

universitätbonn

Klassifikation

Danach, was sie modellieren

Danach, wie sie es modellieren

Danach, wo sie meist eingesetzt werden

Klassifikation: „Was“ und „Wie“

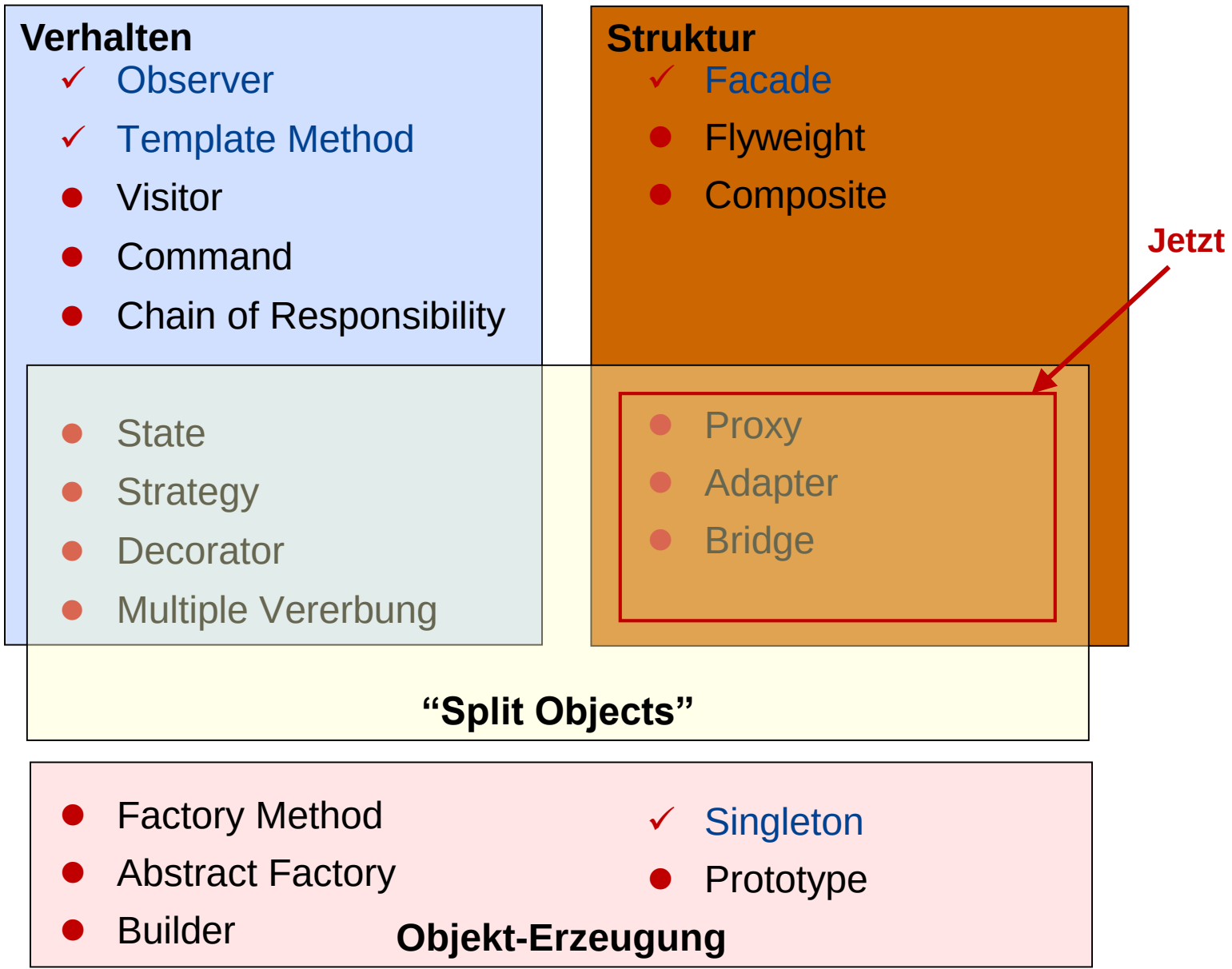
Was?

- Verhaltens-Patterns
  - Verhalten leicht erweiterbar, komponierbar, dynamisch änderbar oder explizit manipulierbar machen
- Struktur-Patterns
  - Zustandslose Objekte (Flyweight), rekursive Strukturen (Composite)
  - Verschiedenste Formen der Entkopplung (Schnittstelle / Implementierung, Identität / physikalische Speicherung, ...)
- Erzeugungs-Patterns
  - Festlegung von konkret zu erzeugenden Objekte (new XYZ()) so weit wie möglich verzögern

Wie?

- Split Object Patterns
  - Ziel: Dynamisch änderbares Verhalten, gemeinsame Verwendung oder Entkopplung von Teilobjekten
  - Weg: Aufteilung eines konzeptionellen Gesamtobjektes in modellierte Teilobjekte die kooperieren um das Verhalten des konzeptionellen Gesamtobjektes zu realisieren

Klassifikation der "Gang of Four"-Patterns (Gamma, Helm, Johnson & Vlissides)



universitätbonn

Wichtige Entwurfsmuster, Teil 1

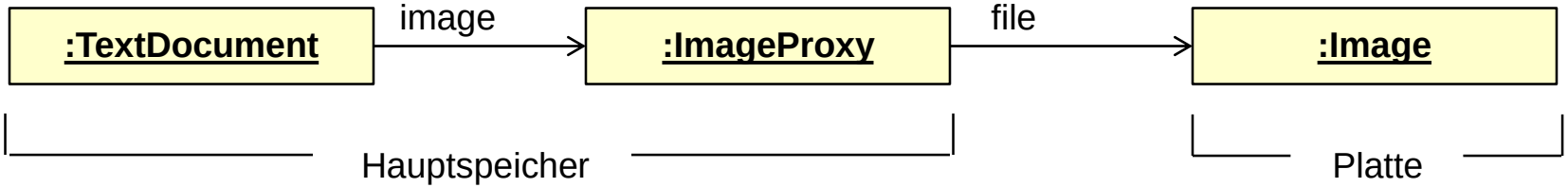
- ✓ Facade
- ✓ Singleton
- → Proxy
- → Adapter
- → Bridge
- → Factory Method
- → Abstract Factory

Oft schon im Systementwurf genutzt

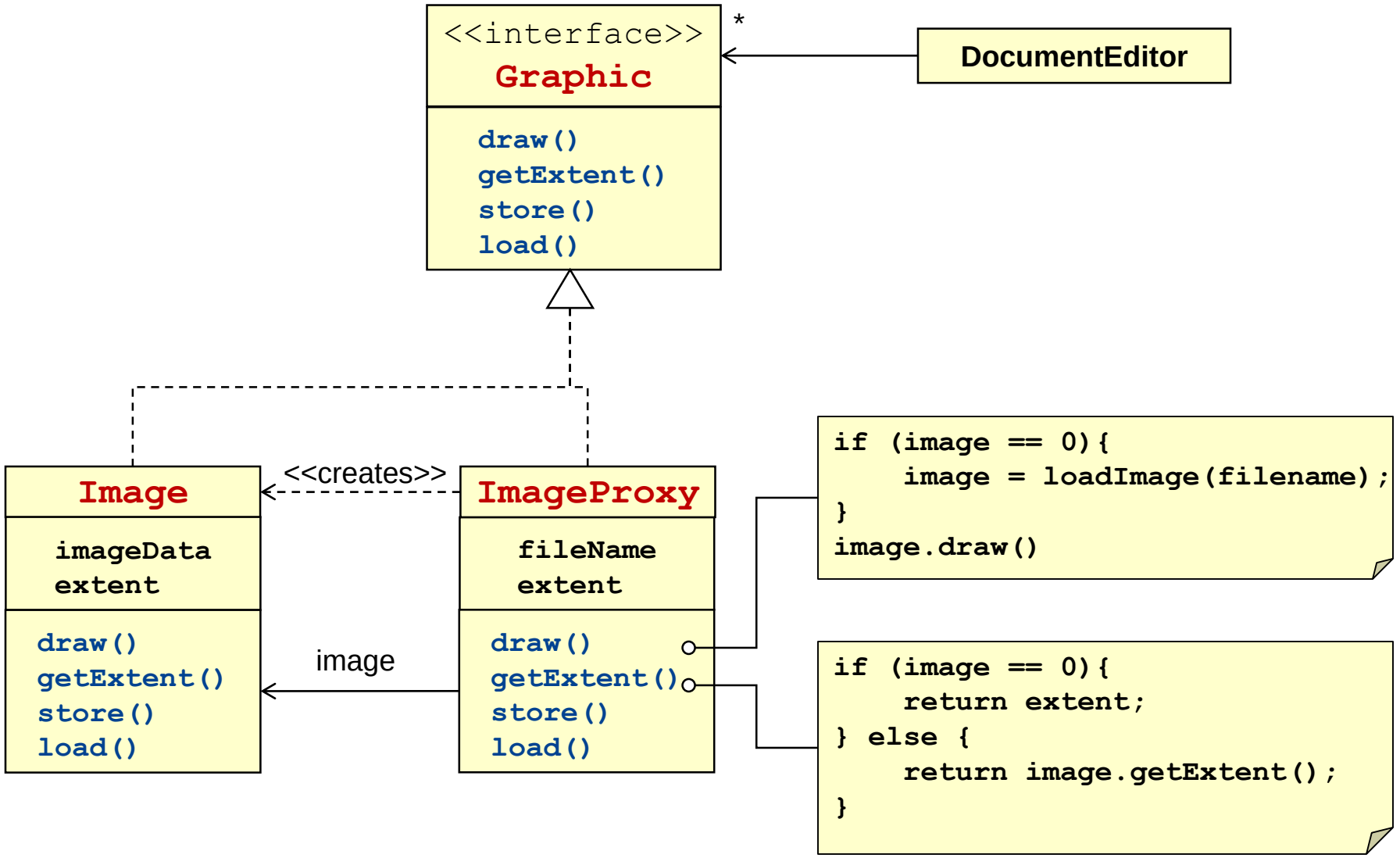
Das Proxy Pattern

Proxy Pattern (auch: Surogate, Smart Reference)

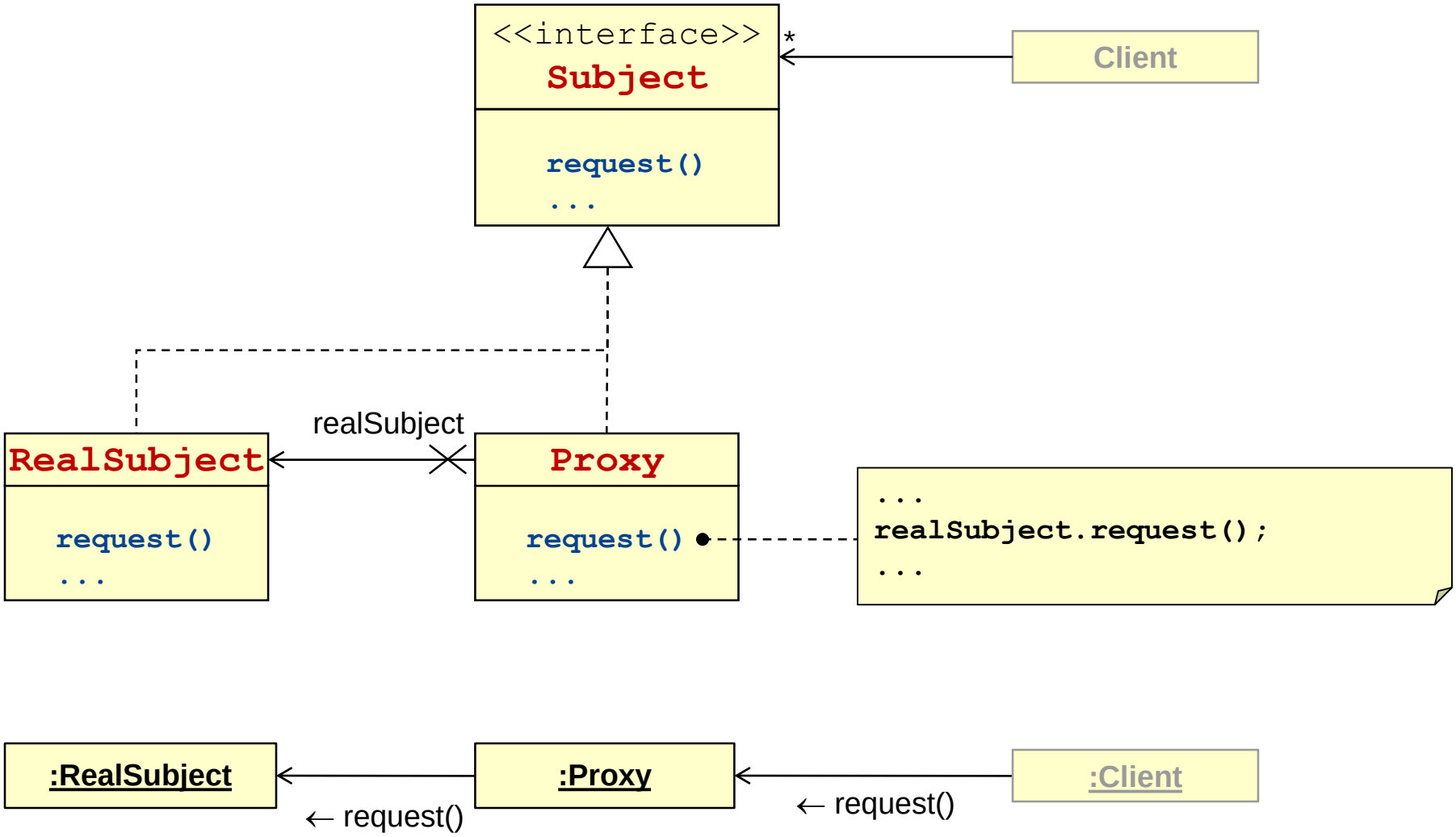
- Absicht
  - Stellvertreter für ein anderes Objekt
  - bietet Kontrolle über Objekt-Erzeugung und -Zugriff
- Motivation
  - kostspielige Objekt-Erzeugung verzögern (zB: große Bilder)
  - verzögerte Objekterzeugung soll Programmstruktur nicht global verändern
- Idee
  - Bild-Stellvertreter (Proxy) verwenden
  - Bild-Stellvertreter verhält sich aus Client-Sicht wie Bild
  - Bild-Stellvertreter erzeugt Bild bei Bedarf



Proxy Pattern: Beispiel



Proxy Pattern: Schema



Proxy Pattern: Verantwortlichkeiten

- Proxy
  - bietet gleiches Interface wie "Subject"
  - referenziert eine "RealSubject"-Instanz
  - kontrolliert alle Aktionen auf dem "RealSubject"
- Subject
  - definiert das gemeinsame Interface
- RealSubject
  - das Objekt das der Proxy vertritt
  - eigentliche Funktionalität
- Zusammenspiel
  - selektives Forwarding

Proxy Pattern: Anwendbarkeit

- Virtueller Proxy
  - verzögerte Erzeugung "teurer" Objekte bei Bedarf
  - Beispiel: Bilder in Dokument, persistente Objekte
- Remote Proxy
  - Zugriff auf entferntes Objekt
  - Objekt-Migration
  - Beispiele: CORBA, RMI, mobile Agenten
- Concurrency Proxy
  - nur eine direkte Referenz auf RealSubject
  - locking des RealSubjects vor Zugriff (threads)
- Copy-on-Write
  - kopieren erhöht nur internen "CopyCounter"
  - wirkliche Kopie bei Schreibzugriff und "CopyCounter">0
  - ➢ Verzögerung teurer Kopier-Operationen

Proxy Pattern: Implementierung der Weiterleitung

„Consultation“ = Weiterleiten von Anfragen

- C++: Operator-Overloading
  - Proxy redefiniert Dereferenzierungs-Operator:*anImage
  - Proxy redefiniert Member-Accesss-Operator:anImage->extent()
- Smalltalk: Reflektion
  - Proxy redefiniert Methode "doesNotUnderstand: aMessage"
- Java: Reflektion
  - Dynamic Proxy implementiert „InvocationHandler“ Interface
- Lava: Schlüsselwort
  - class Proxy { private consultee RealImage ri; …}
- Kotlin: Schlüsselwort
  - class Proxy(ri: RealImage) : RealImage by ri {... }

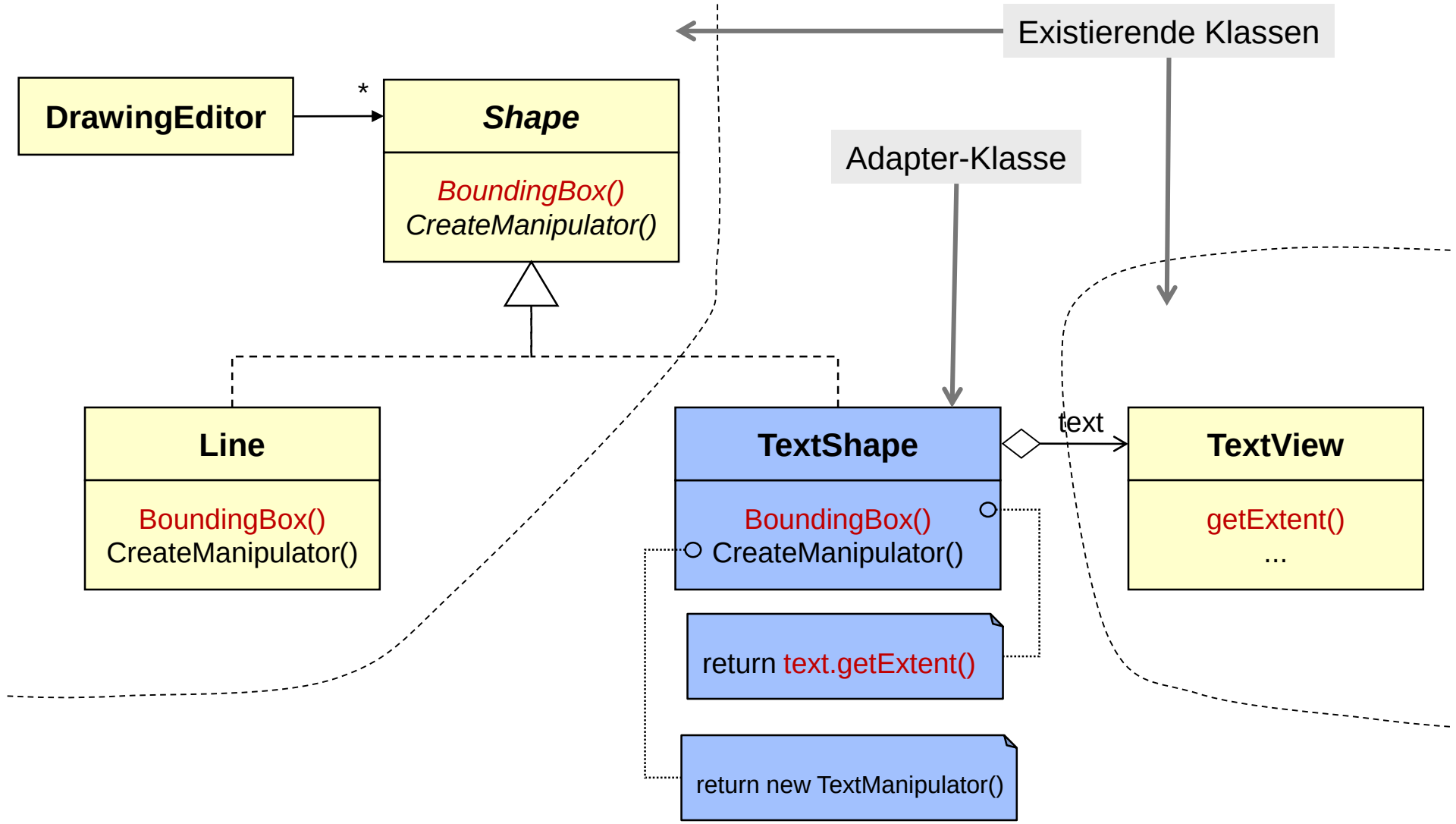
Für alle in Proxy nicht definierten Methoden aus RealImage werden vom Compiler in Proxy Methoden generiert, die an das Feld ri weiterleiten

Das Adapter Pattern

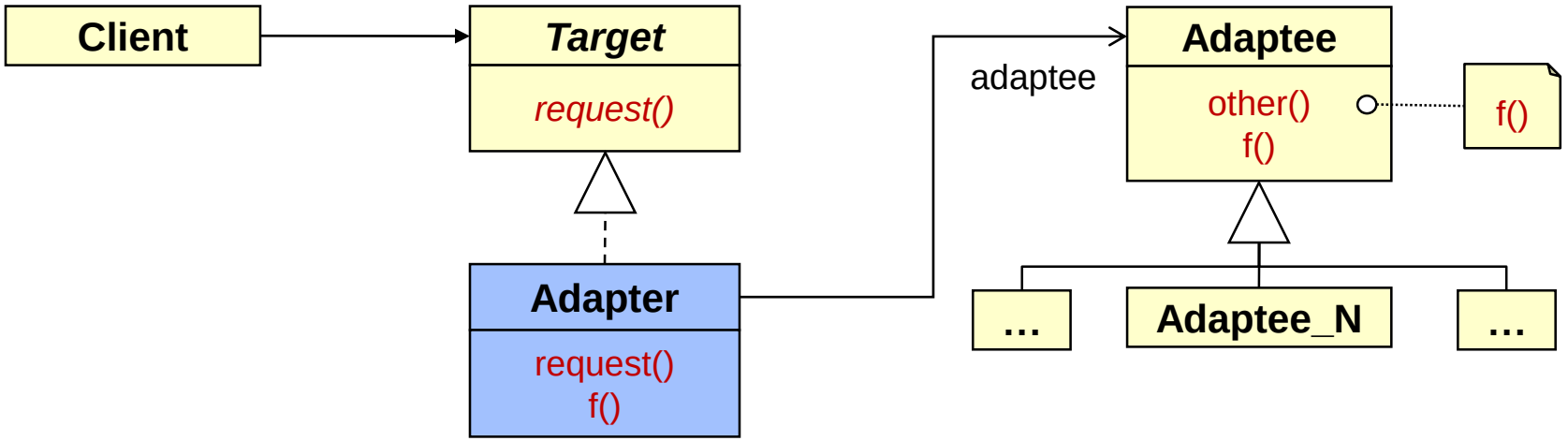
Adapter Pattern (auch: Wrapper)

- Absicht
  - Schnittstelle existierender Klasse an Bedarf existierender Clients anpassen
- Motivation
  - Graphik-Editor bearbeitet Shapes
    - ⇨ Linien, Rechtecke, ...
    - ⇨ durch "BoundingBox" beschrieben
  - Textelemente sollen auch möglich sein
    - ⇨ Klasse TextView vorhanden
    - ⇨ bietet keine "BoundingBox"-Operation
  - Integration ohne
    - ⇨ Änderung der gesamten Shape-Hierarchie und ihrer Clients
    - ⇨ Änderung der TextView-Klasse
- Idee
  - Adapter-Klasse stellt Shape-Interface zur Verfügung
  - ... implementiert Shape-Interface anhand der verfügbaren TextView-Methoden

Adapter Pattern: Beispiel



Adapter Pattern (Objekt-Adapter): Schema



Adaptiert eine ganze Klassenhierarchie

- Beliebige Adapter-Subtypen können mit beliebigen Adaptee-Subtypen kombiniert werden
- Kombination wird erst zur Laufzeit, auf Objektebene, festgelegt

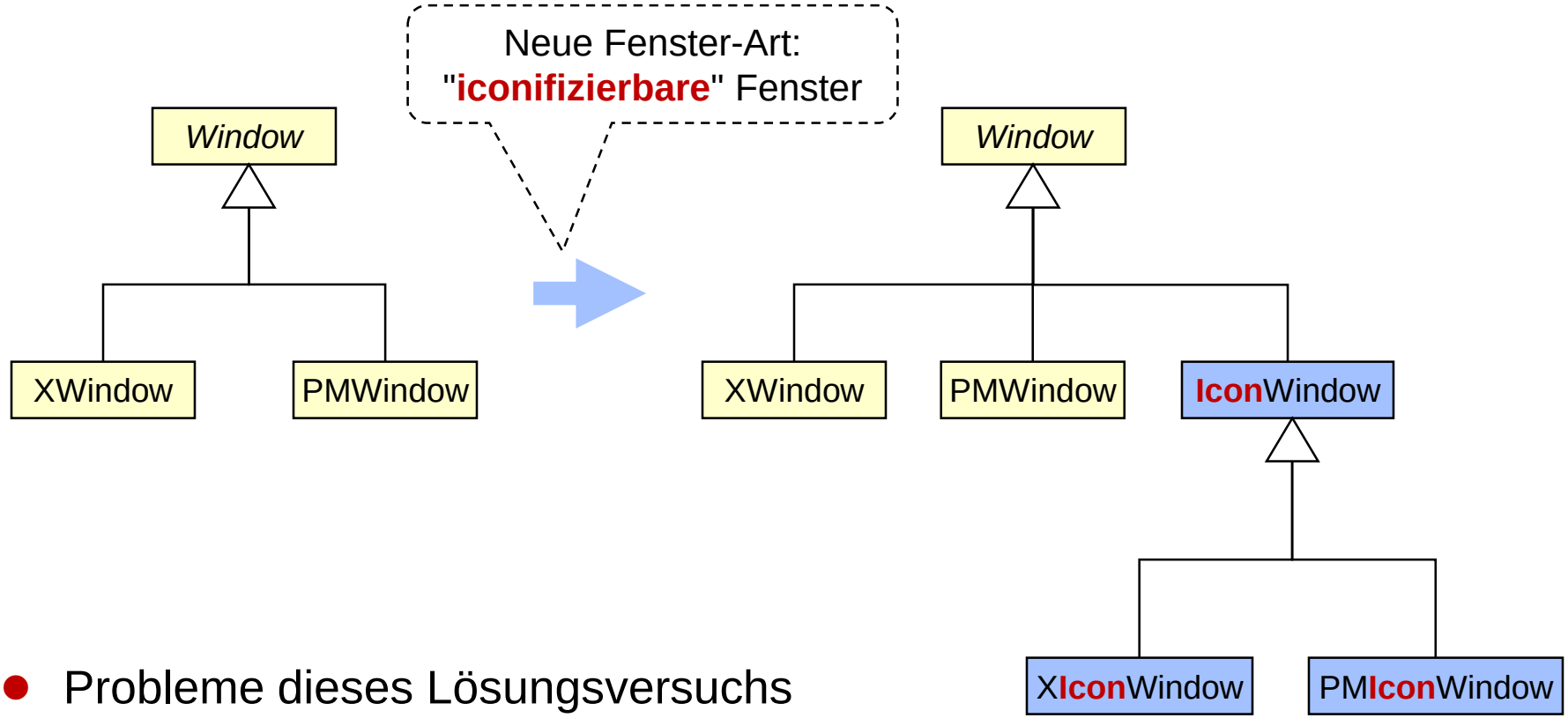
Das Bridge Pattern

Bridge Pattern (auch: Handle / Body)

- Absicht
  - Schnittstelle und Implementierung trennen
  - ... unabhängig variieren

- Motivation
  - portable "Window"-Abstraktion in GUI-Toolkit
  - mehrere Variations-Dimensionen
    - ⇨ Fenster-Arten:
      - normal / als Icon,
      - schließbar / nicht schließbar,
      - ...
    - ⇨ Implementierungen:
      - X-Windows,
      - IBM Presentation Manager,
      - MacOS,
      - Windows XYZ,
      - ...

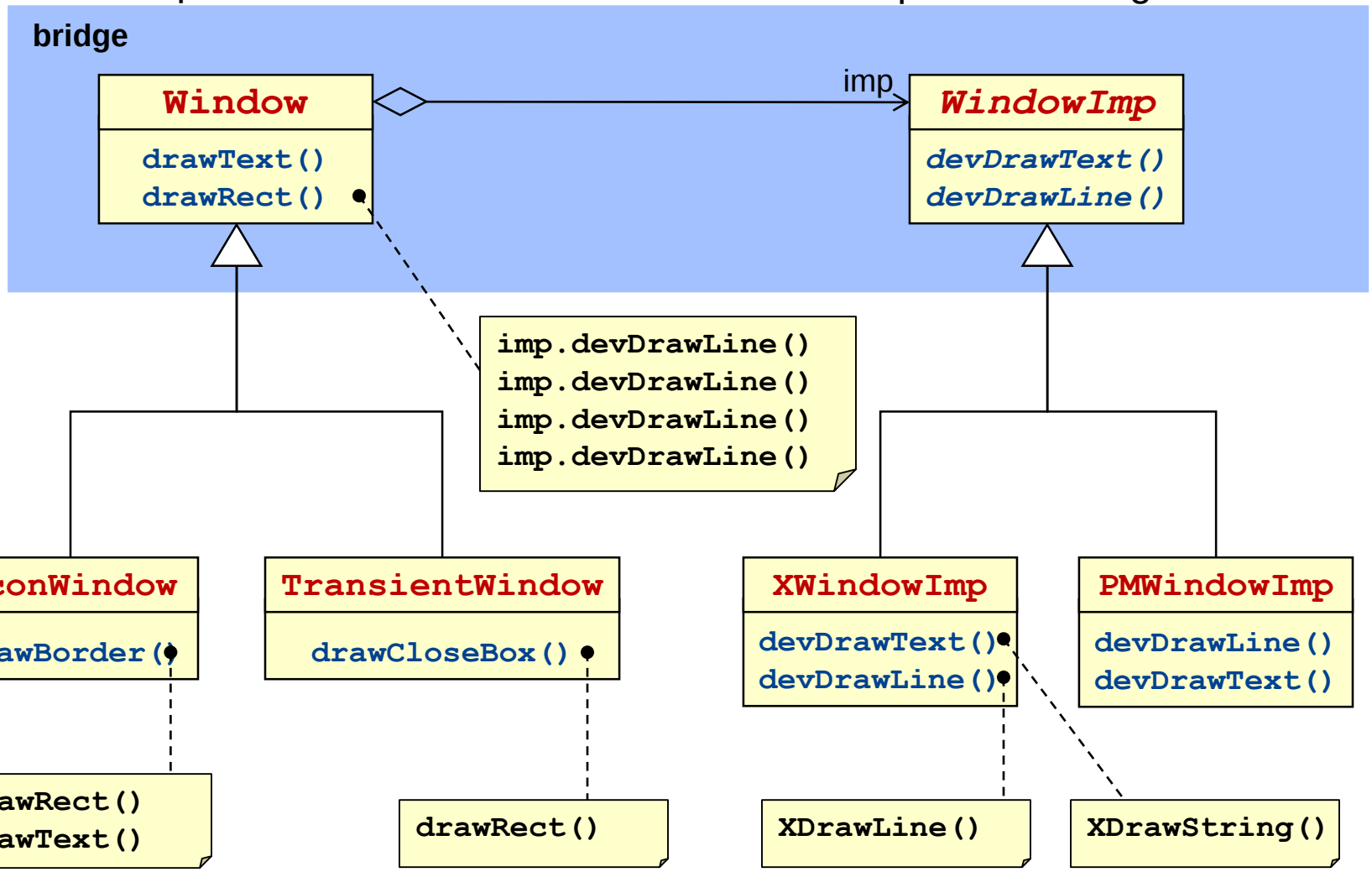
Bridge Pattern: Warum nicht einfach Vererbung einsetzen?



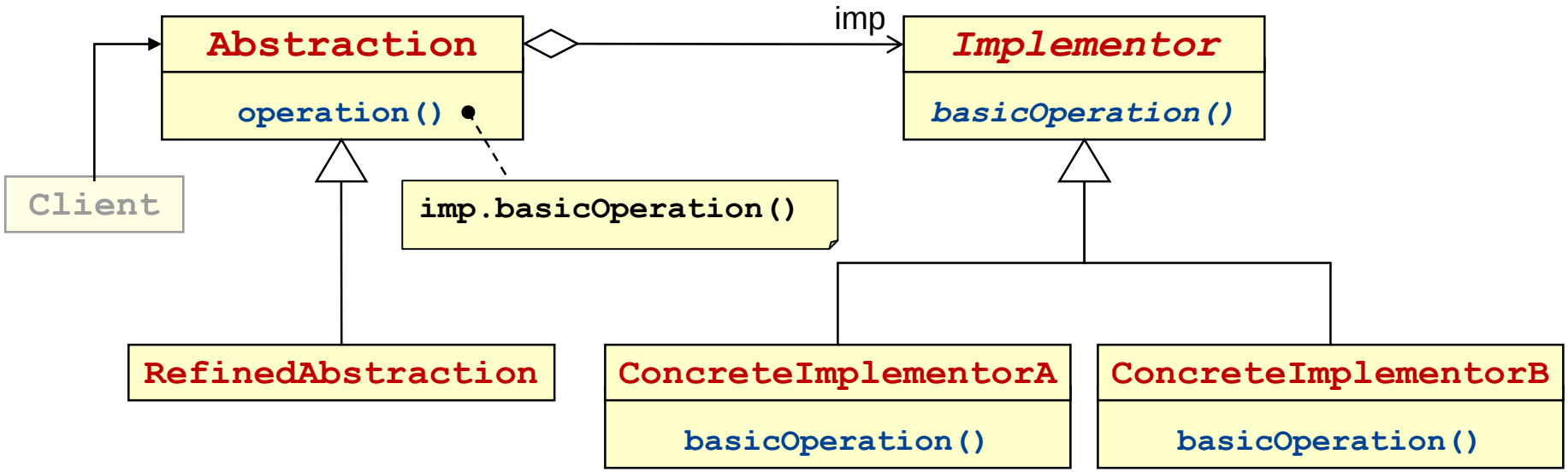
- Probleme dieses Lösungsversuchs
  - Eigene Klasse für jede Kombination Fenster-Art / Plattform
    - ⇨ unwartbar
  - Client wählt Kombination Fenster-Art / Plattform (bei der Objekterzeugung)
    - ⇨ plattformabhängiger Client-Code

Bridge Pattern: Idee

- Trennung von
  - Konzeptueller Hierarchie
  - Implementierungs-Hierarchie



Bridge Pattern: Schema

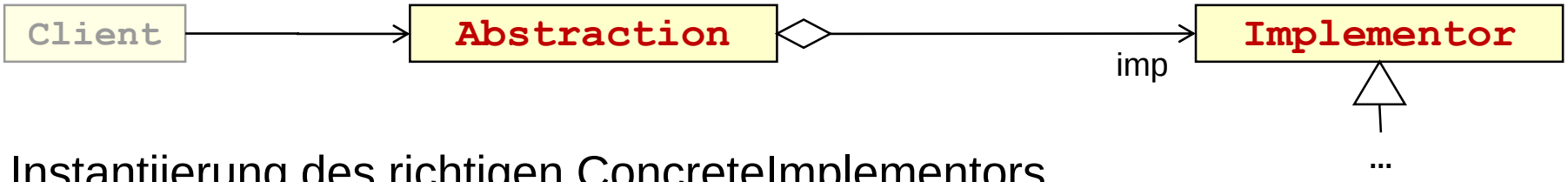


- Abstraction
  - definiert Interface
  - referenziert Implementierung
- RefinedAbstraction
  - erweitert das Interface
- Implementor
  - Interface der Implementierungs-Hierarchie
  - kann von Abstraction abweichen
- ConcreteImplementor
  - wirkliche Implementierung

Bridge Pattern: Anwendbarkeit

- Dynamische Änderbarkeit
  - Implementierung erst zur Laufzeit auswählen
- Unabhängige Variabilität
  - neue Unterklassen in beiden Hierarchien beliebig kombinierbar
- Implementierungs-Transparenz für Clients
  - Änderungen der Implementierung erfordern keine Änderung / Neuübersetzung der Clients
- Vermeidung von "Nested Generalisations"
  - keine Hierarchien der Art wie in der Motivations-Folie gezeigt
  - keine kombinatorische Explosion der Klassenanzahl
- Sharing
  - mehrere Clients nutzen gleiche Implementierung
  - z.B. Strings

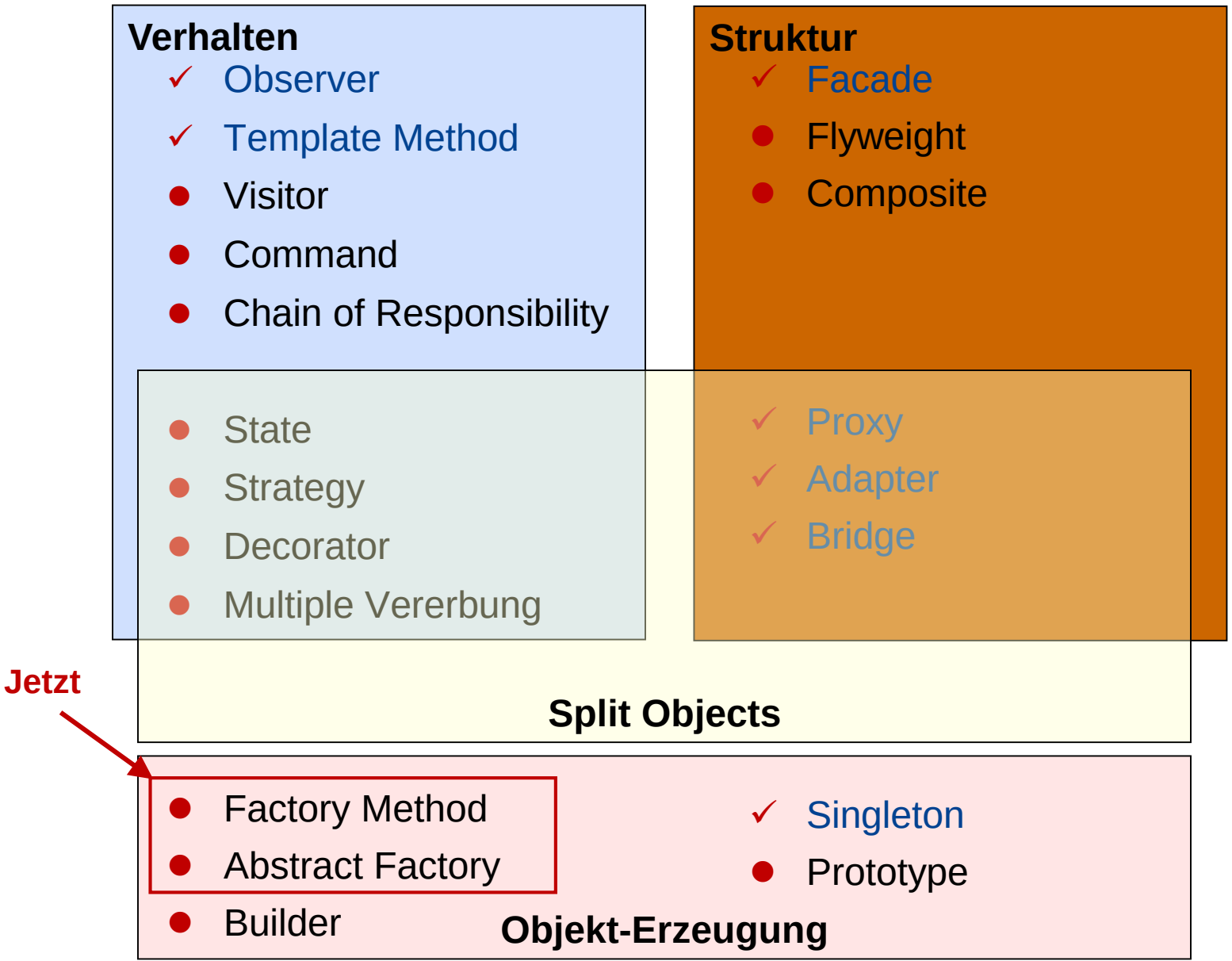
Bridge Pattern: Implementierung



Instantiierung des richtigen ConcreteImplementors

- Falls Abstraction alle ConcreteImplementor-Klassen kennt:
  - Fallunterscheidung im Konstruktor der ConcreteAbstraction
  - Auswahl des ConcreteImplementor anhand von Parametern des Konstruktors
  - Alternativ: Default-Initialisierung und spätere situationsbedingte Änderung
- Falls Abstraction völlig unabhängig von ConcreteImplementor-Klassen sein soll:
  - Entscheidung anderem Objekt überlassen
  - Abstract Factory Pattern

Zwischenstand

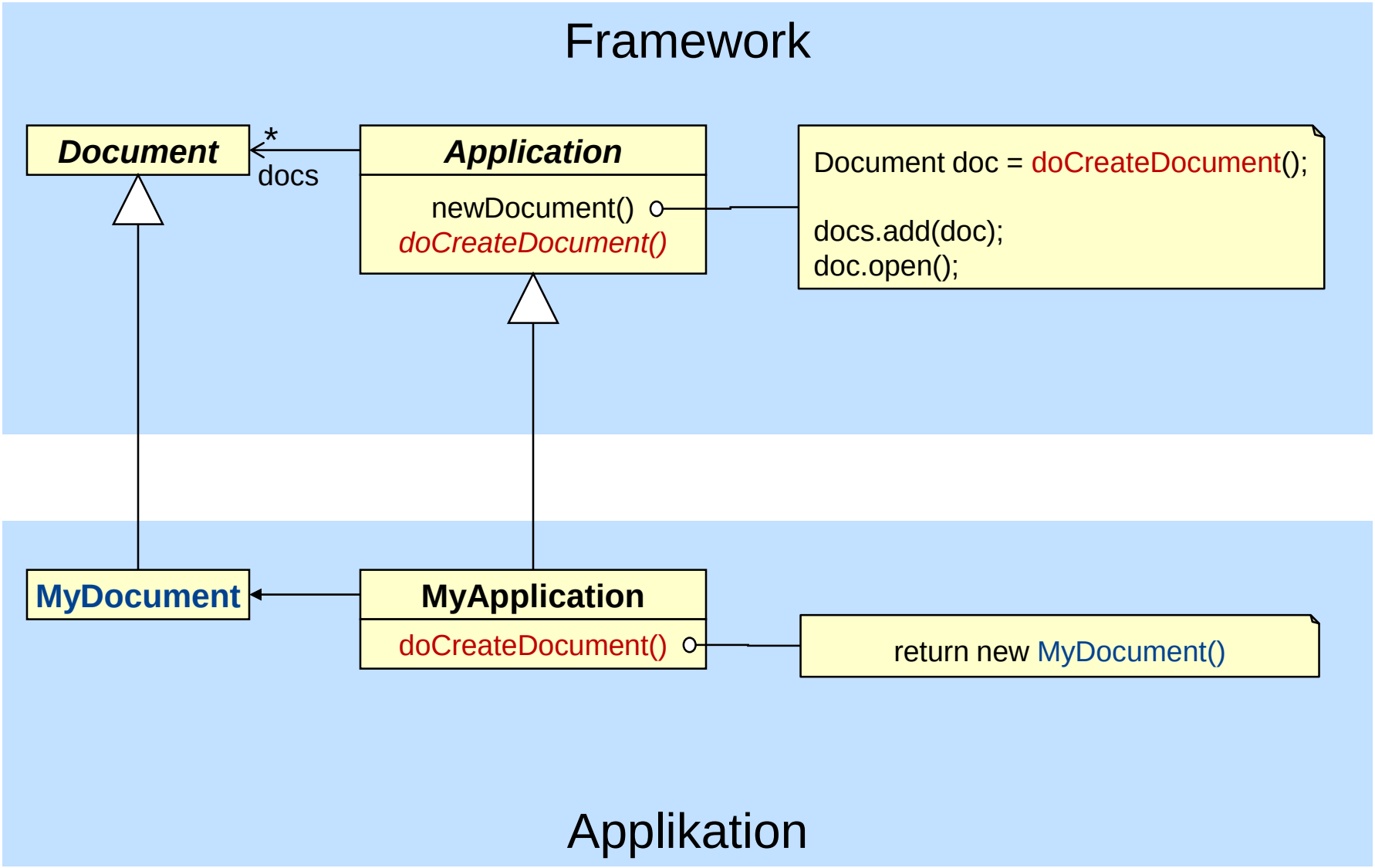


Das Factory Method Pattern

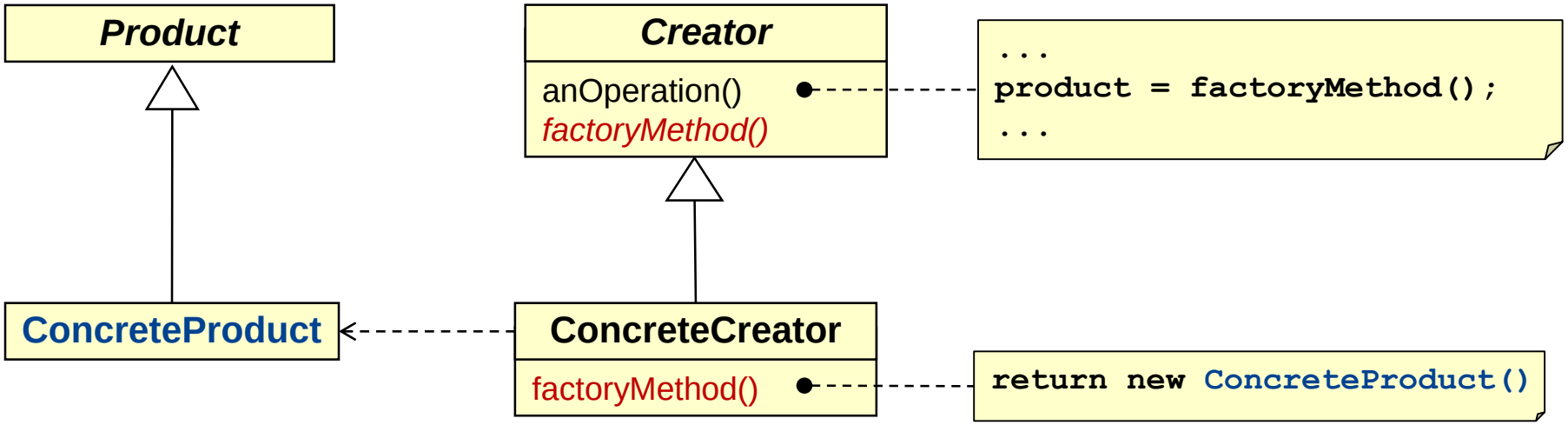
Factory Method

- Ziel
  - Entscheidung über konkreter Klasse neuer Objekte verzögern
- Motivation
  - Framework mit vordefinierten Klassen "Document" und "Application"
  - Template-Methode, für das Öffnen eines Dokuments:
    - ⇨ 1. Check ob Dokument-Art bekannt
    - ⇨ 2. Erzeugung einer Document-Instanz !?!
  - Erzeugung von Instanzen noch nicht bekannter Klassen!
- Idee
  - aktuelle Klasse entscheidet wann Objekte erzeugt werden
    - ⇨ Aufruf einer abstrakten Methode
  - Subklasse entscheidet was für Objekte erzeugt werden
    - ⇨ implementiert abstrakte Methode passend

Factory Method: Beispiel



Factory Method: Schema und Rollen

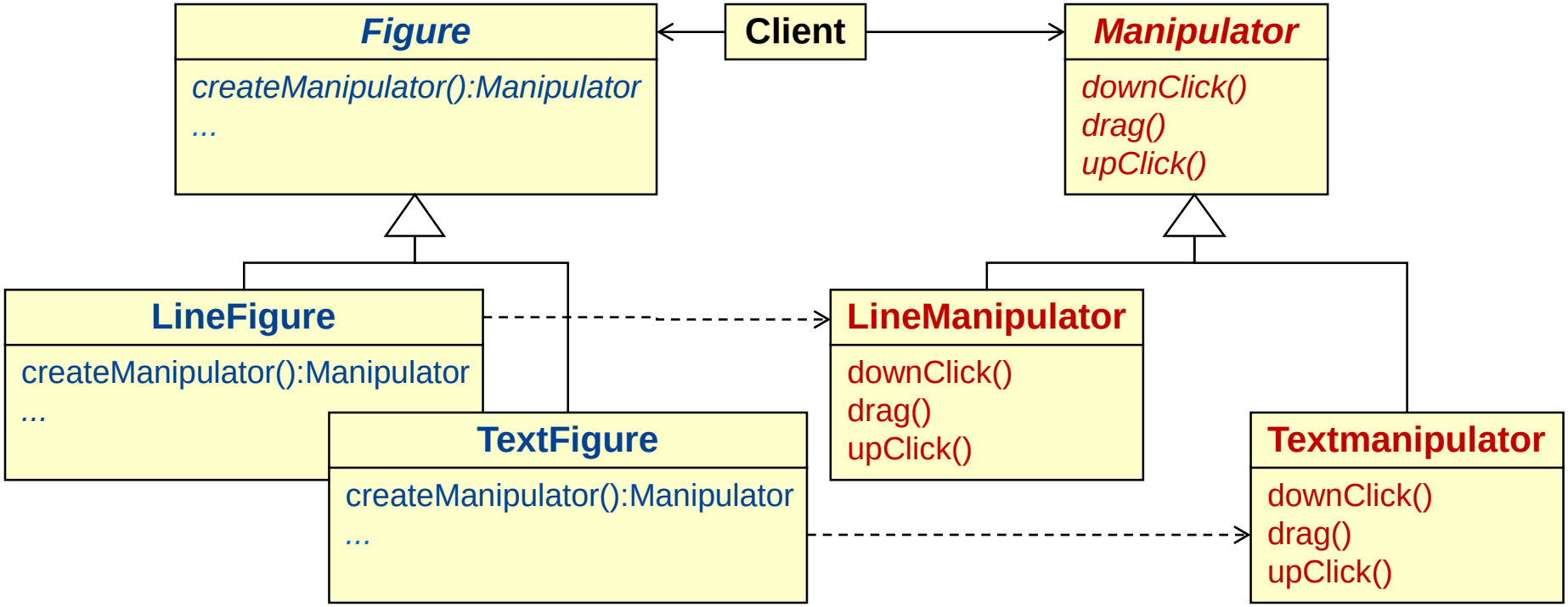


- Factory Method
  - kann abstrakt sein (s. Diagramm) oder Default-Implementierung haben
- Creator
  - Klasse im Framework, die die Factory Method (abstrakt oder als „hook“) definiert
- Concrete Creator
  - Klasse in der Applikation, die die Factory Method konkret (re)definiert

Factory Method: Anwendbarkeit

- Klasse neuer Objekte unbekannt
  - Eine Klasse (Creator) muss Objekte erzeugen, deren Art sie nicht vorhersehen kann
- Mehrere Hilfsklassen
  - Wissen über konkrete Hilfsklassen an einer Stelle lokalisieren
  - Beispiel: Parallele Hierarchien (nächste Folie)
- Verzögerte Entscheidung über Art der erzeugten Objekte
  - erst in Subklasse (ConcreteCreator)
  - erst beim Aufruf (durch Parametrisierung – siehe Implementierungsvarianten)

Factory Method: Anwendbarkeit



- Verbindung paralleler Klassenhierarchien
  - Factory Method lokalisiert das Wissen über zusammengehörige Klassen
  - Restlicher Code der Figure-Hierarchie und Client-Code ist völlig unabhängig von *spezifischen* Manipulators (nur vom Manipulator-Interface)

Factory Method: Implementierung
1. Allgemein

- Fester "Produkt-Typ"
  - Basisvariante: jede Unterklasse erzeugt festgelegten Produkt-Typ
  - Vorteil: Inkrementelle Erweiterbarkeit
  - Nachteil: Für jede neue Produkt-Art muss eine neue Unterklasse erzeugt werden

```
class Creator {
  Product create(){ MyProduct(); }
}
```

- Codierter "Produkt-Typ"
  - Parameter codiert den "Produkt-Typ"
  - Fallunterscheidung anhand Code
  - Vorteil: nur eine Methode und Klasse nötig (keine Unterklassen)
  - Nachteil: Änderung der Methode, für jeden neuen Produkttyp (keine inkrementelle Erweiterbarkeit)

```
class Creator {
  Product create(int id) {
    if (id==1) return MyProduct1();
    if (id==2) return MyProduct2();
    ...
  }
}
```

Frage: Geht auch beides?

Nur eine Methode/Klasse, aber trotzdem keine Änderung für neue Produkttypen?

Factory Method: Implementierung 2. Sprachspezifisch

- Instanz der Klasse „Class“ als "Produkt-Typ"
  - „Produkt-Typ“ ist ein (Klassen-) Objekt, kein primitiver Datentyp
  - Generische Instanz-Erzeugung durch Nachricht an das Klassenobjekt
  - Generische Lösung aber
  - … kein statischer Typ-Check
  - … nur in reflektiven Sprachen (Smalltalk, Java, …)

```
class Creator {
  Object create(Class prodType) {
    return prodType.newInstance();
  }
}
```

Reflektiver Aufruf des parameterlosen Default-Konstruktors in Java

Factory Method: Konsequenzen

- Code wird abstrakter / wiederverwendbarer
  - keine festen Abhängigkeiten von spezifischen Klassen
- Verbindung paralleler Klassenhierarchien
  - Factory Method lokalisiert das Wissen über Zusammengehörigkeiten
- evtl. künstliche Subklassen
  - nur zwecks Objekterzeugung

Factory Method: Anwendungen

- überall in Toolkits, Frameworks, Class Libraries
  - Unidraw: Beispiel "Figure und Manipulator"
  - MacApp und ET++: Beispiel "Document und Application"
- Smalltalk's Model-View-Controller Framework
  - FactoryMethoden-Template: `defaultController`
  - Hook-Methode: `defaultControllerClass`
- Orbix
  - Erzeugung des richtigen Proxy
    - ⇨ leichte Ersetzung von Standard-Proxy durch Caching Proxy

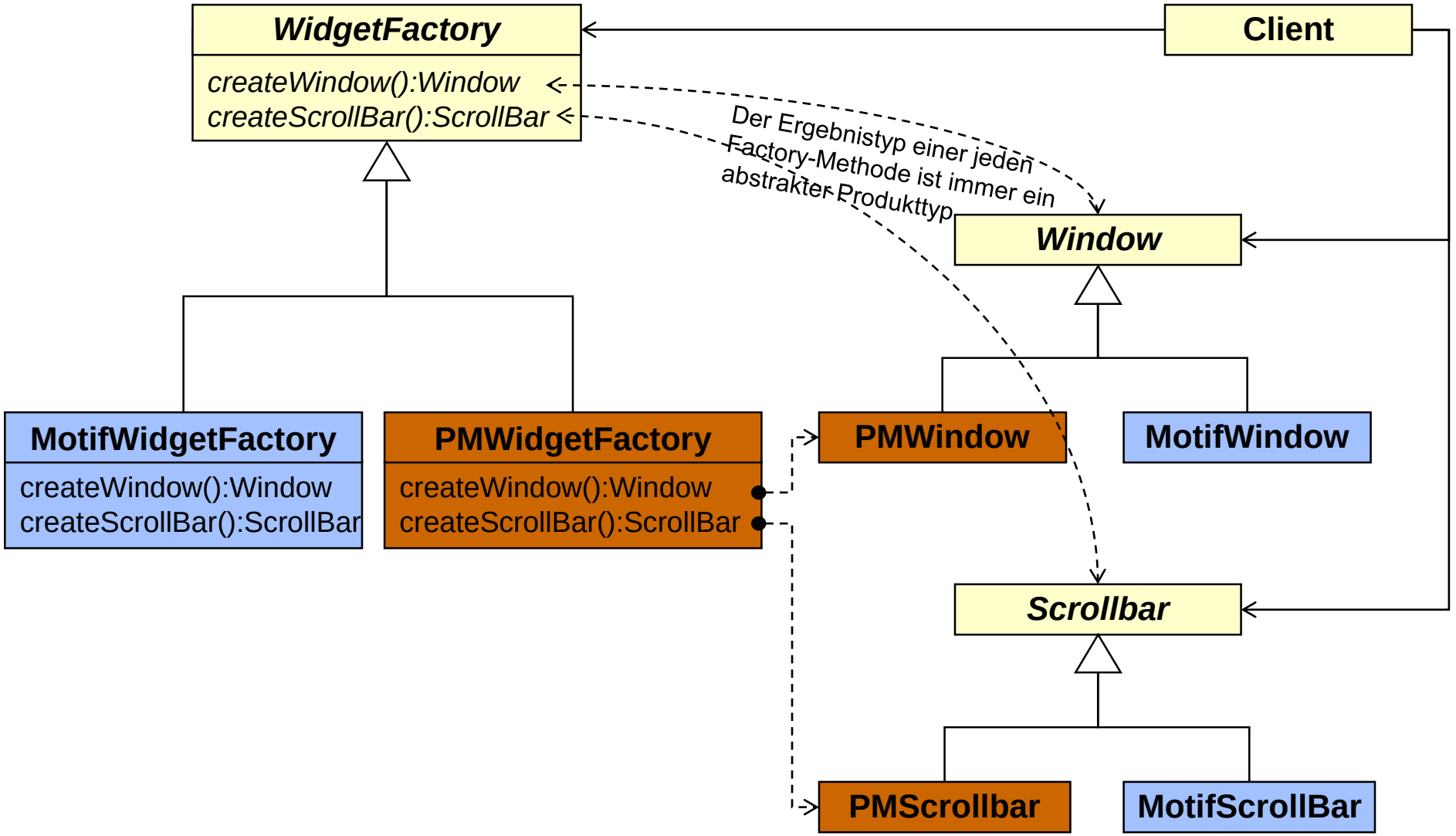
Das Abstract Factory Pattern

Abstract Factory

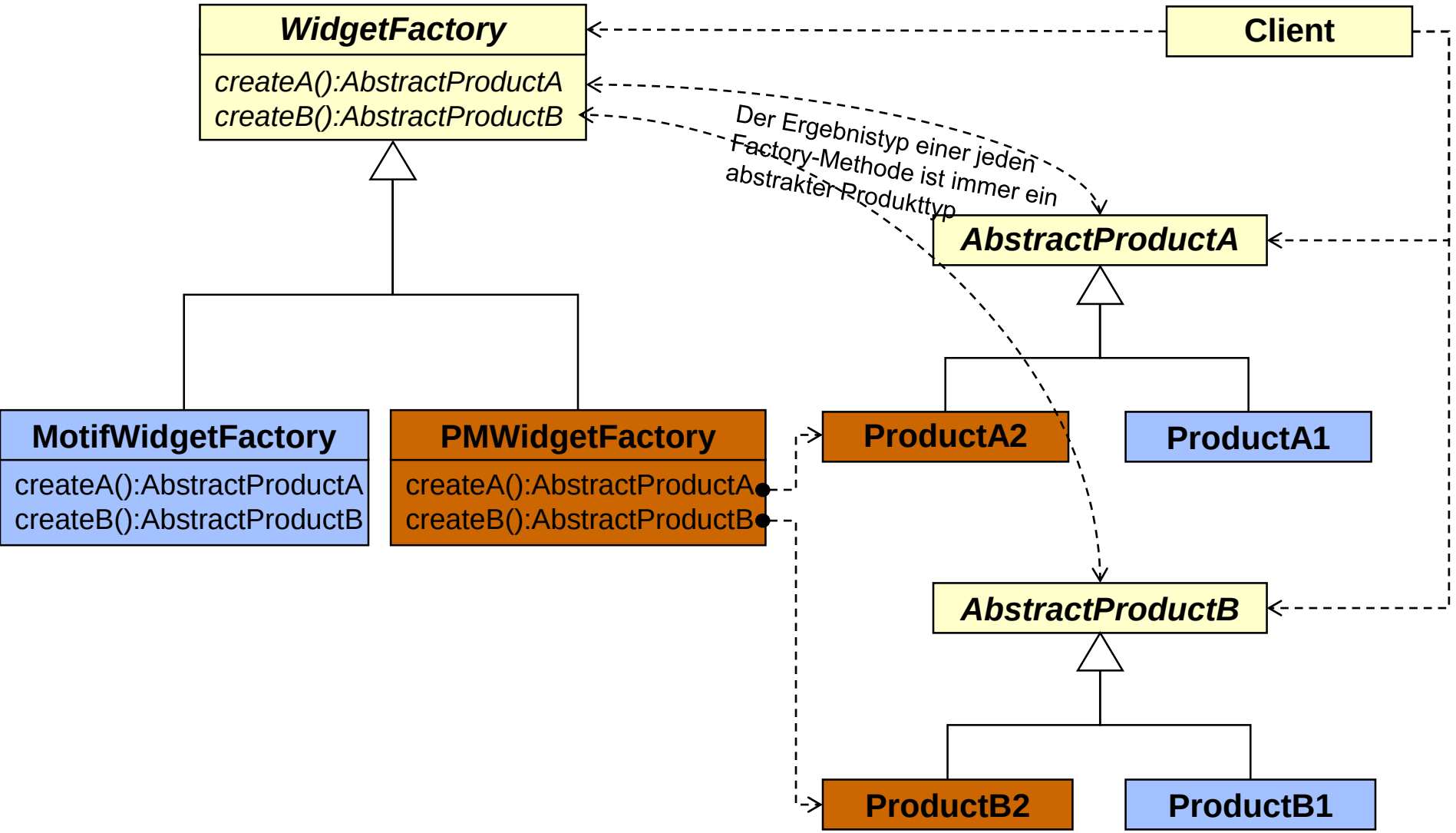
- Ziel
  - zusammengehörige Objekte verwandter Typen erzeugen
  - ... ohne deren Klassenzugehörigkeit fest zu codieren

- Motivation
  - GUI-Toolkit für mehrere Plattformen
  - Anwendungsklassen nicht von plattformspezifischen Widgets abhängig machen
  - Trotzdem soll die Anwendung
    - ⇨ alle Widgets konsistent zu einem "look and feel" erzeugen können
    - ⇨ "look and feel" umschalten können

Abstract Factory: Beispiel



Abstract Factory: Schema



Abstract Factory: Implementierung

- ConcreteFactories sind Singletons
- Produkt-Erzeugung via Factory-Methods
- fester Produkt-Typ (eine Methode für jedes Produkt)
  - Nachteil
    - ⇨ neues Produkt erfordert Änderung der gesamten Factory-Hierarchie
- Codierter "Produkt-Typ" (1 parametrisierte Methode für alle Produkte)
  - Vorteil
    - ⇨ leichte Erweiterbarkeit um neues Produkt
  - Nachteile:
    - ⇨ alle Produkte müssen gemeinsamen Obertyp haben (im schlimmsten Fall „Object“)
    - ⇨ Clients können nur die Operationen dieses Typs nutzen
    - ⇨ Verzicht auf statische Typsicherheit wenn man von dort „herunter-casten“ muss
- Klassen-Objekt als "Produkt-Typ" (eine parametrisierte Methode)
  - Vorteil
    - ⇨ neues Produkt erfordert keinerlei Änderungen der Factory
  - Nachteil
    - ⇨ Verzicht auf jegliche statische Typinformation / Typsicherheit

Abstract Factory: Implementierung

- Produktfamilie = Subklasse
  - Vorteil
    - ⇨ Konsistenz der Produkte
  - Nachteil
    - ⇨ neue Produktfamilie erfordert neue Subklasse, auch bei geringen Unterschieden

- Produktfamilie = von Clients konfigurierte assoziative Datenstruktur
  - Varianten
    - ⇨ Prototypen und Cloning
    - ⇨ Klassen und Instantiierung
  - Vorteil
    - ⇨ keine Subklassenbildung erforderlich
  - Nachteil
    - ⇨ Verantwortlichkeit an Clients abgegeben
    - ⇨ konsistente Produktfamilien können nicht mehr garantiert werden

Abstract Factory: Konsequenzen

- Abschirmung von Implementierungs-Klassen
  - Namen von Implementierungsklassen erscheinen nicht im Code von Clients
  - Clients benutzen nur abstraktes Interface
- Leichte Austauschbarkeit von Produktfamilien
  - Name einer ConcreteFactory erscheint nur ein mal im Code
    - ⇨ bei ihrer Instantiierung
  - Leicht austauschbar gegen andere ConcreteFactory
  - Beispiel: Dynamisches Ändern des look-and-feel
    - ⇨ andere ConcreteFactory instantiieren
    - ⇨ alle GUI-Objekte neu erzeugen
- Konsistente Benutzung von Produktfamilien
  - Keine Motif-Scrollbar in Macintosh-Fenster
- Schwierige Erweiterung um neue Produktarten
  - Schnittstelle der AbstractFactory legt Produktarten fest
  - Neue Produktart = Änderung der gesamten Factory-Hierarchie

Abstract Factory: Anwendbarkeit

- System soll unabhängig sein von
  - Objekt-Erzeugung
  - Objekt-Komposition
  - Objekt-Darstellung
- System soll konfigurierbar sein
  - Auswahl aus verschiedenen Produkt-Familien
- konsistente Produktfamilien
  - nur Objekte der gleichen Familie "passen zusammen"
- Library mit Produktfamilie anbieten
  - Clients sollen nur Schnittstelle kennen
  - ... nicht die genauen Teilprodukt-Arten / -Implementierungen

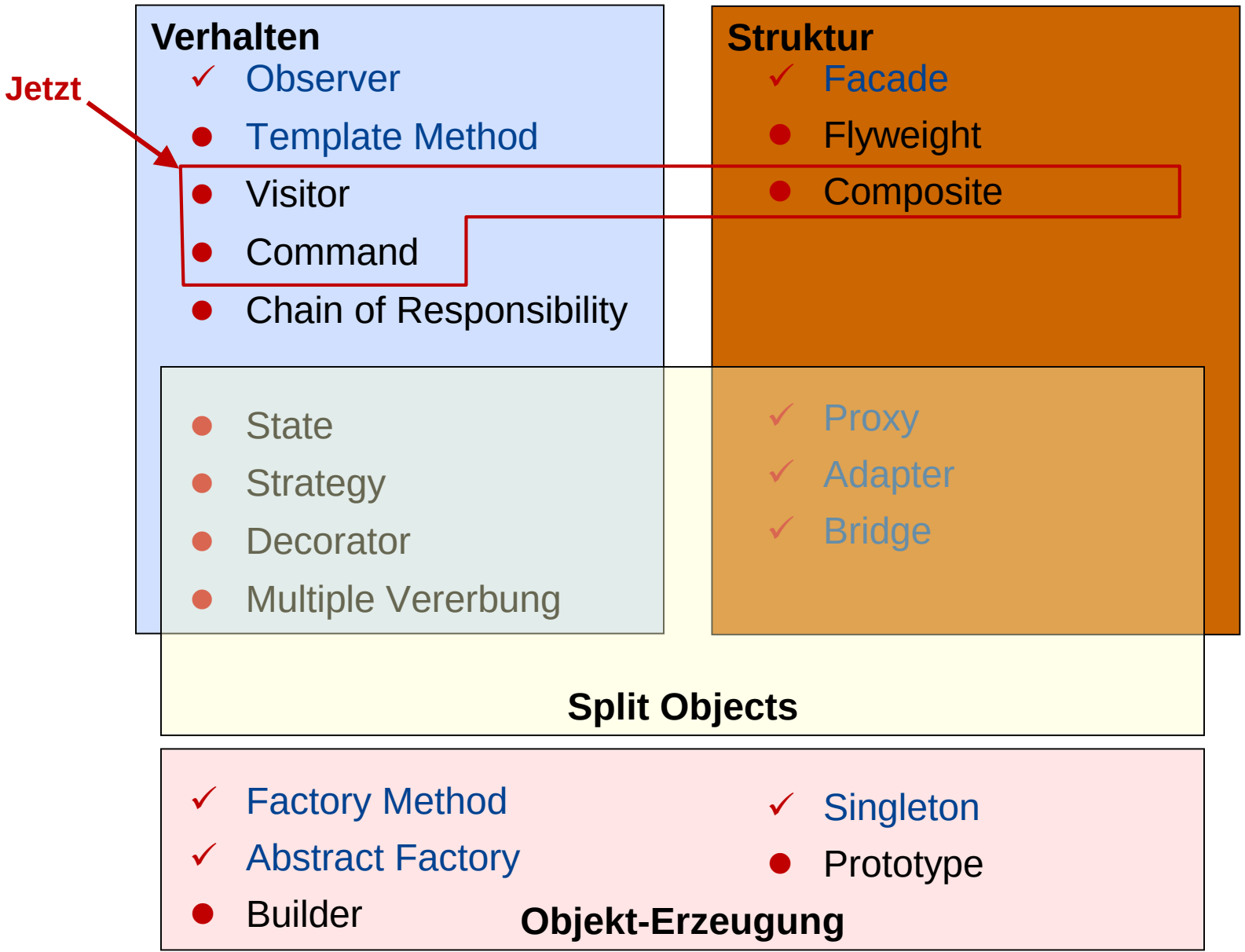
universitätbonn

Wichtige Entwurfsmuster, Teil 2

Command
Composite
Visitor

} Typischerweise im Objektentwurf genutzt

Zwischenstand



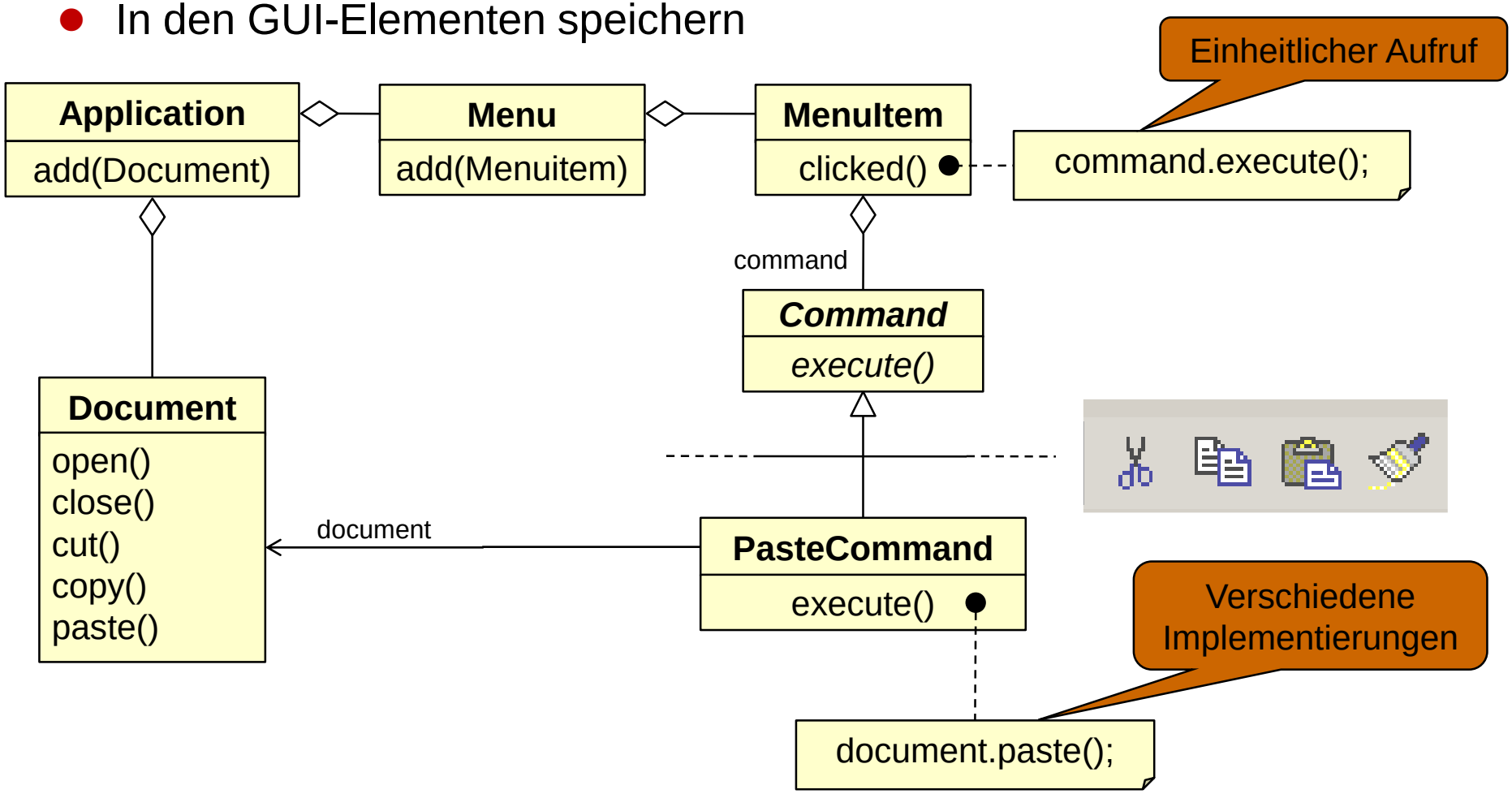
Das Command Pattern

Das Command Pattern: Motivation

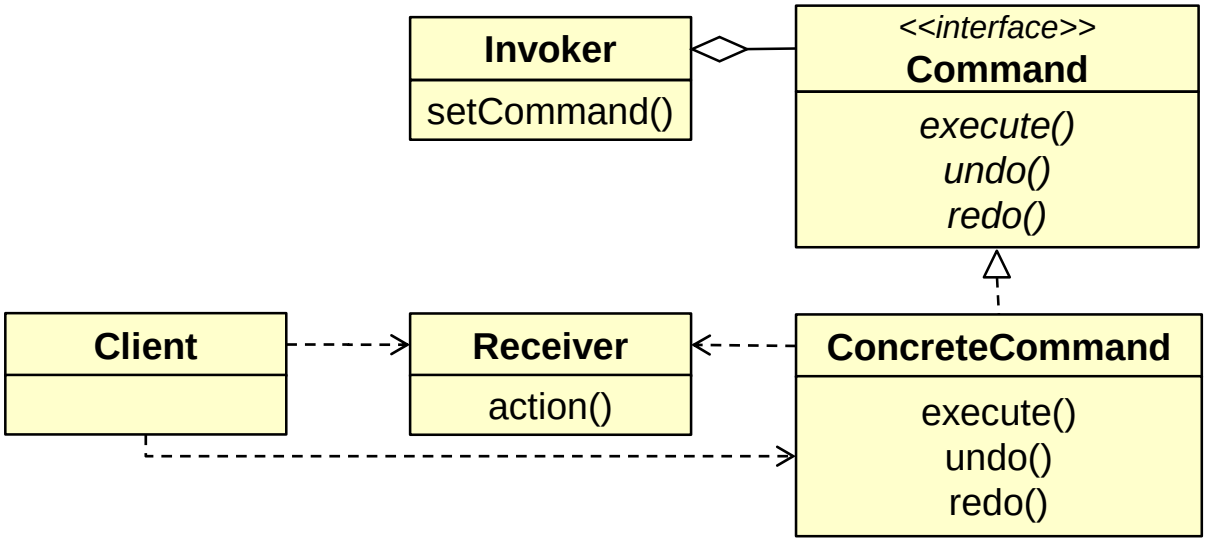
- Kontext
  - GUI-Toolkits mit Buttons, Menüs, etc.
- Forces
  - Wiederverwendung
    - ⇨ Über verschiedene GUI-Elemente ansprechbare Operationen nur ein mal programmieren
  - Dynamik
    - ⇨ dynamische Änderung der Aktionen von Menu-Einträgen
    - ⇨ kontext-sensitive Aktionen
  - Vielfalt trotz Einheitlichkeit
    - ⇨ Einheitlicher Code in allen GUI-Elementen
    - ⇨ .. trotzdem verschiedene Effekte

Das Command Pattern: Idee

- Operationen als Objekte mit Methode execute() darstellen
- In den GUI-Elementen speichern



Command Pattern: Schema und Rollen



- Command Interface
  - legt fest was Commands tun können
  - Mindestens Execute, optional auch Undo, Redo
- ConcreteCommand
  - Implementiert Interface
  - „Controllers“ sind oft „ConcreteCommands“

- Execute-Methode
  - Ausführen von Kommandos
- Undo-Methode
  - Rückgängig machen von Kommandos
- Redo-Methode
  - Rückgängig gemachtes Kommando wieder ausführen

Das Command Pattern: Konsequenzen

- Entkopplung
  - von Aufruf einer Operation und Spezifikation einer Operation.
- Kommandos als Objekte
  - Command-Objekte können wie andere Objekte auch manipuliert und erweitert werden.
- Komposition
  - Folgen von Command-Objekte können zu weiteren Command-Objekten zusammengefasst werden → Makros!
- Erweiterbarkeit
  - Neue Command-Objekte können leicht hinzugefügt werden, da keine Klassen geändert werden müssen.

Das Command Pattern: Anwendbarkeit

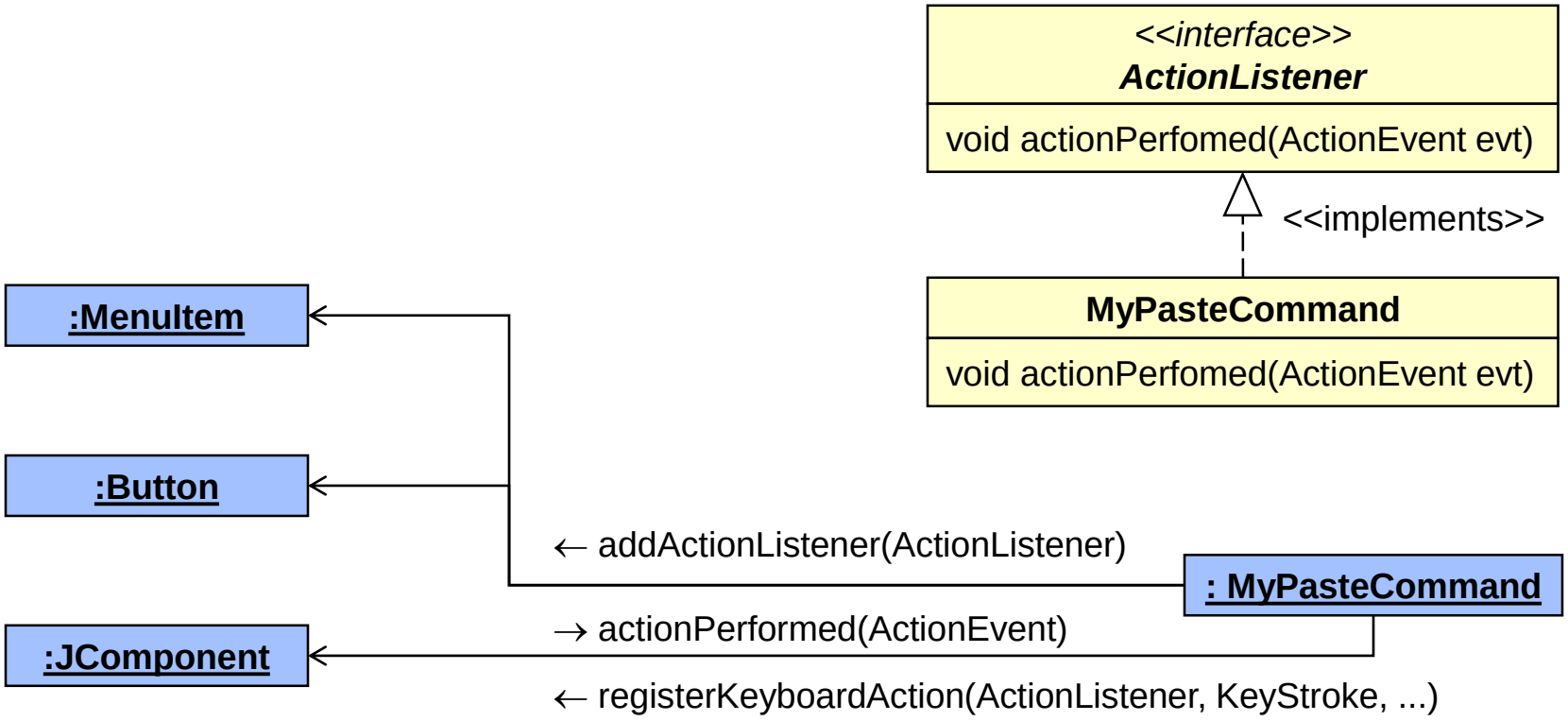
- Operationen als Parameter
- Variable Aktionen
  - referenziertes Command-Objekt austauschen
- Zeitliche Trennung
  - Befehl zur Ausführung, Protokollierung, Ausführung
- Speicherung und Protokollierung von Operationen
  - History-Liste
  - Serialisierung
- "Undo" von Operationen
  - Command-Objekte enthalten neben execute() auch unexecute()
  - werden in einer History-Liste gespeichert
- Recover-Funktion nach Systemabstürzen
  - History-Liste wird gespeichert
  - Zustand eines Systems ab letzter Speicherung wiederstellbar
- Unterstützung von Transaktionen
  - Transaktionen können sowohl einfache ("primitive"), als auch komplexe Command-Objekte sein.

Implementierung des Command Patterns

- Unterschiedliche Grade von Intelligenz
  - Command-Objekte können "nur" Verbindung zwischen Sender und Empfänger sein, oder aber alles selbstständig erledigen.
- Unterstützung von "undo"
  - Semantik
    - ⇨ Undo (unexecute()) und redo (execute()) müssen den exakt gegenteiligen Effekt haben!
  - Problem: In den wenigsten Fällen gibt es exact inverse Operationen
    - ⇨ Die Mathematik ist da eine Ausnahme...
  - Daher Zusatzinfos erforderlich
    - ⇨ Damit ein Command-Objekt "undo" unterstützen kann, müssen evtl. zusätzliche Informationen gespeichert werden.
    - ⇨ Typischerweise: Kopien des alten Zustands der Objekte die wiederhergestellt werden sollen.
  - Tiefes Klonen
    - ⇨ Es reicht nicht eine Referenz auf die Objekte zu setzen!
    - ⇨ Man muss das Objekt "tief" klonen, um sicherzustellen dass sein Zustand nicht verändert wird.
  - History-Liste
    - ⇨ Für mehrere Levels von undo/redo

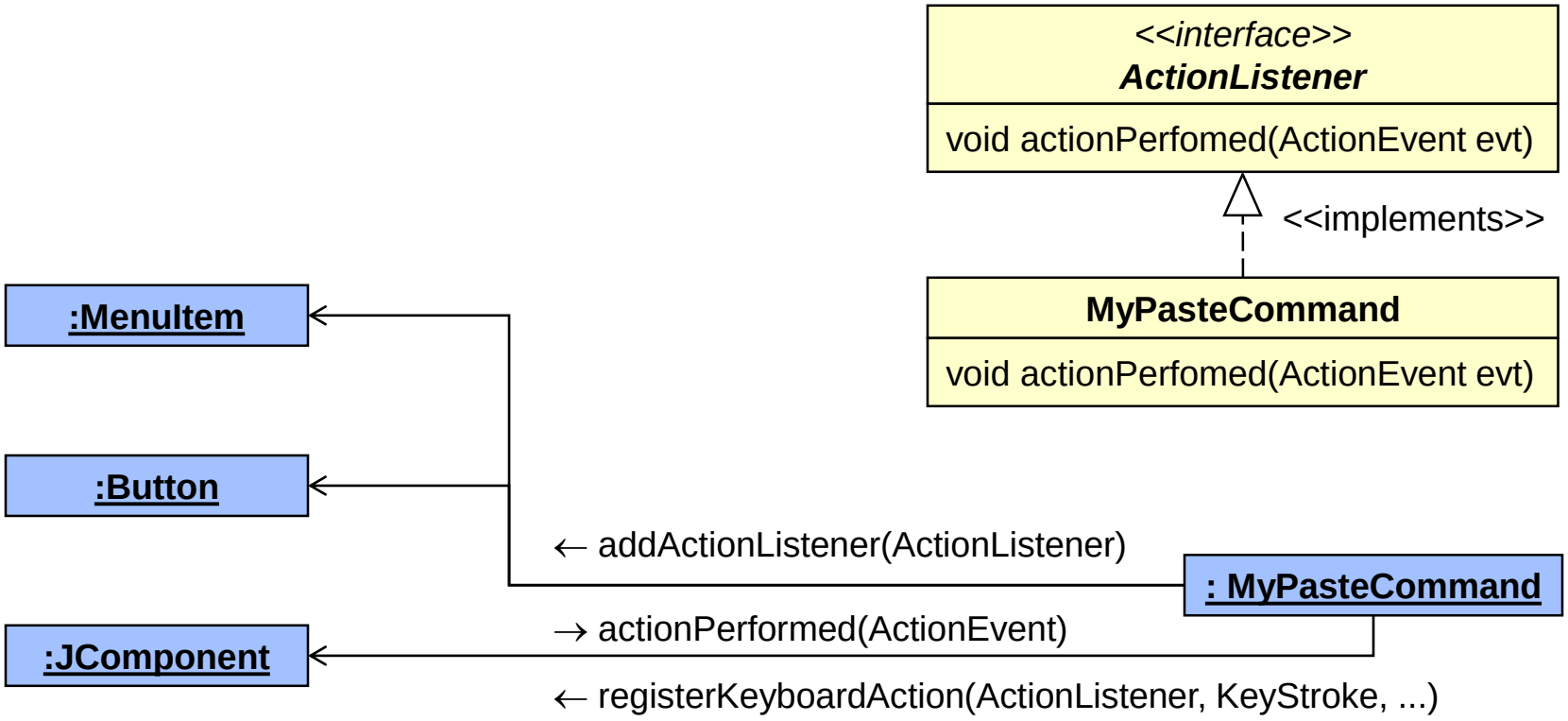
Verbindung von Command und Observer im JDK

Verbindung von Observer und Command in Java



- Observer (= ActionListener)
  - Buttons, Menue-Einträge und Tasten generieren "ActionEvents"
  - Interface "ActionListener" ist vordefiniert
  - "ActionListener" implementieren
  - ... und Instanzen davon bei Buttons, MenuItems, etc registrieren
- Command & Observer
  - gleiche Methode (z.B. actionPerformed) spielt die Rolle der run() Methode eines Commands und die Rolle der update() Methode eines Observers
  - implementierende Klasse (z.B. MyPasteCommand) ist gleichzeitig ein Command und ein Observer

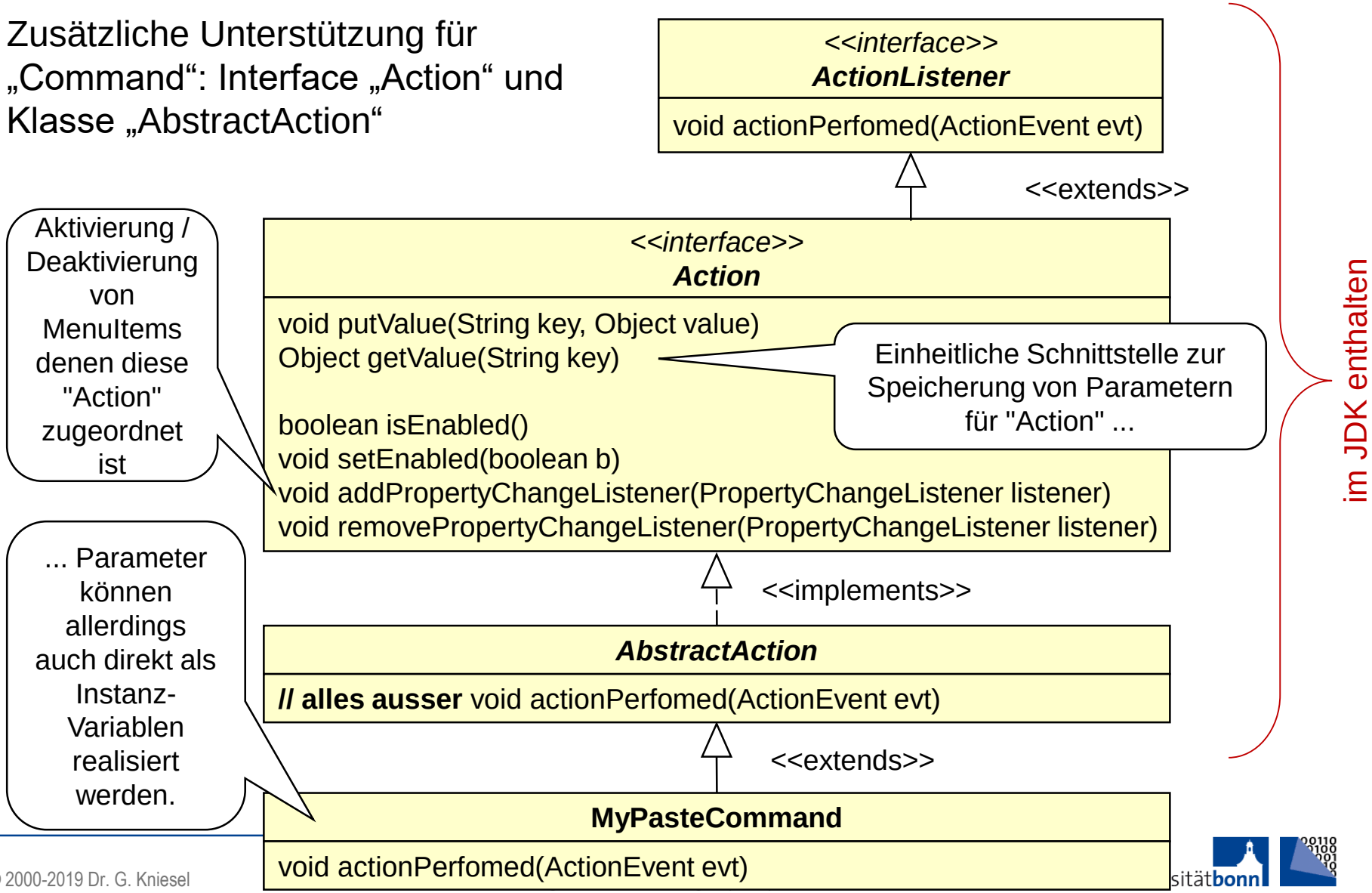
Verbindung von Observer und Command in Java verbindet auch Push und Pull



- Push Observer
  - Subject „pusht“ eine „Event“-Instanz
  - Der „Event“-Typ kann selbst definiert werden
  - In dieser Instanz können somit weitere Informationen mit „gepusht“ werden (als Werte von Instanzvariablen)
- Pull Observer
  - Die als Parameter übergebene „Event“-Instanz enthält immer eine Referenz auf die Quelle des Events
  - Somit ist es möglich von dort weitere Informationen zu anzufragen („pull“)

Verbindung von Observer und Command in Java (2)

- Zusätzliche Unterstützung für „Command“: Interface „Action“ und Klasse „AbstractAction“



Beispiel: Änderung der Hintergrundfarbe (1)

```
class ColorAction extends AbstractAction
{  public ColorAction(..., Color c, Component comp)
   {  ...
      color = c;
      target = comp;
   }

   public void actionPerformed(ActionEvent evt)
   {  target.setBackground(color);
      target.repaint();
   }

   private Component target;
   private Color color;
}
```

```
class ActionButton extends JButton
{  public ActionButton(Action a)
   {  ...
      addActionListener(a);
   }
}
```

- ColorAction
  - Ändern der Hintergrundfarbe einer GUI-Komponente
- ActionButton
  - Buttons die sofort bei Erzeugung mit einer Action verknüpft werden

Beispiel: Änderung der Hintergrundfarbe (2)

- Nutzung der ColorAction
  - Erzeugung einer Instanz
  - Registrierung

```
class SeparateGUIFrame extends JFrame
{ public SeparateGUIFrame()
  {  ...
     JPanel panel = new JPanel();

     Action blueAction = new ColorAction("Blue", ...,..., panel);

     panel.add(new ActionButton(blueAction));

     panel.registerKeyboardAction(blueAction,...);

     JMenu menu = new JMenu("Color");
     menu.add(blueAction);
     ...
  }
}
```

Fazit

- ✓ Wiederverwendung
  - gleiche **`Action`** für Menu, Button, Key
- ✓ Elegante Verbindung von Push- und Pull-Observer und Command
  - in Commands sind **`ActionListener`** von Buttons, Menus, etc.
    - ⇨ Einheitlicher Aufruf via **`actionPerformed(ActionEvent evt)`**
  - Buttons und Menus sind **`PropertyChangeListener`** von Commands
    - ⇨ Aktivierung / Deaktivierung
  - Push- durch Daten im übergebenen Event-Objekt
  - Pull möglich durch Rückreferenz auf das Event-Auslösende Objekt im Event-Objekt

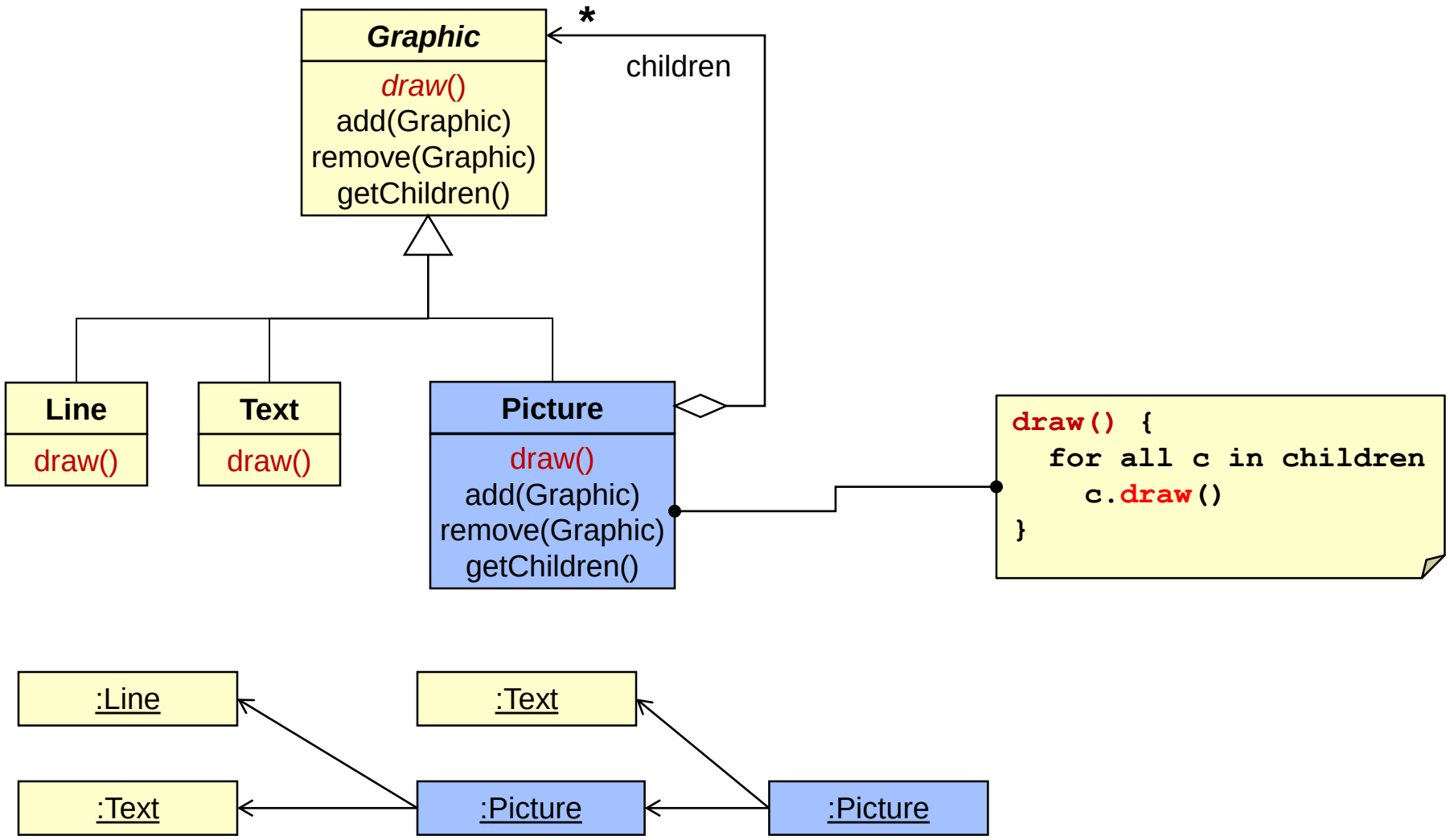
- Dynamik
  - Wie ändert man die aktuelle Aktion?
  - ... konsistent für alle betroffenen MenuItems, Buttons, etc.???
  - ➢ Strategy Pattern! (→ Kapitel „Objektentwurf“)

Das Composite Pattern

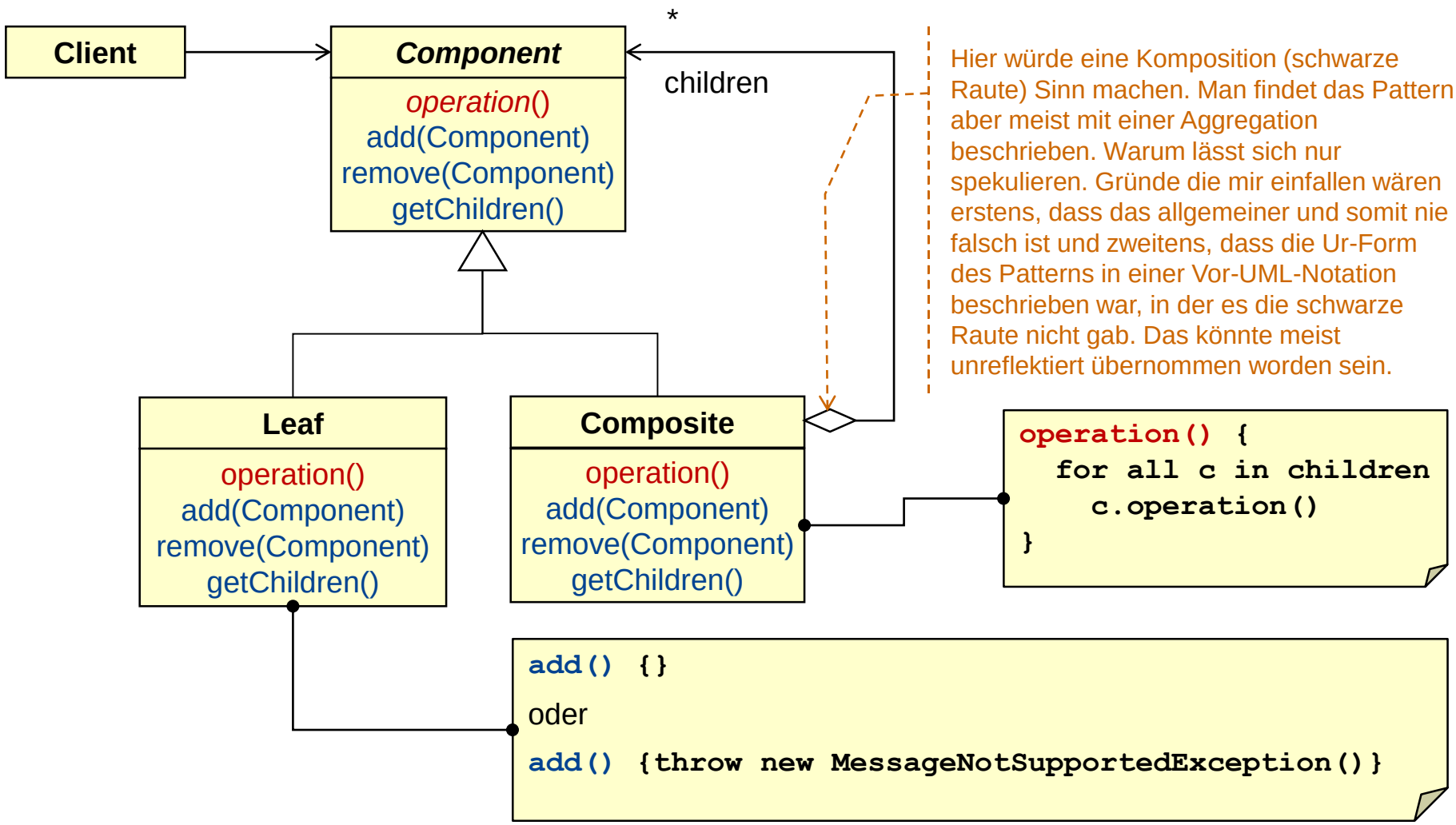
Composite Pattern

- Ziel
  - rekursive Aggregations-Strukturen darstellen ("ist Teil von")
  - Aggregat und Teile einheitlich behandeln
- Motivation
  - Gruppierung von Graphiken

Composite Pattern: Beispiel

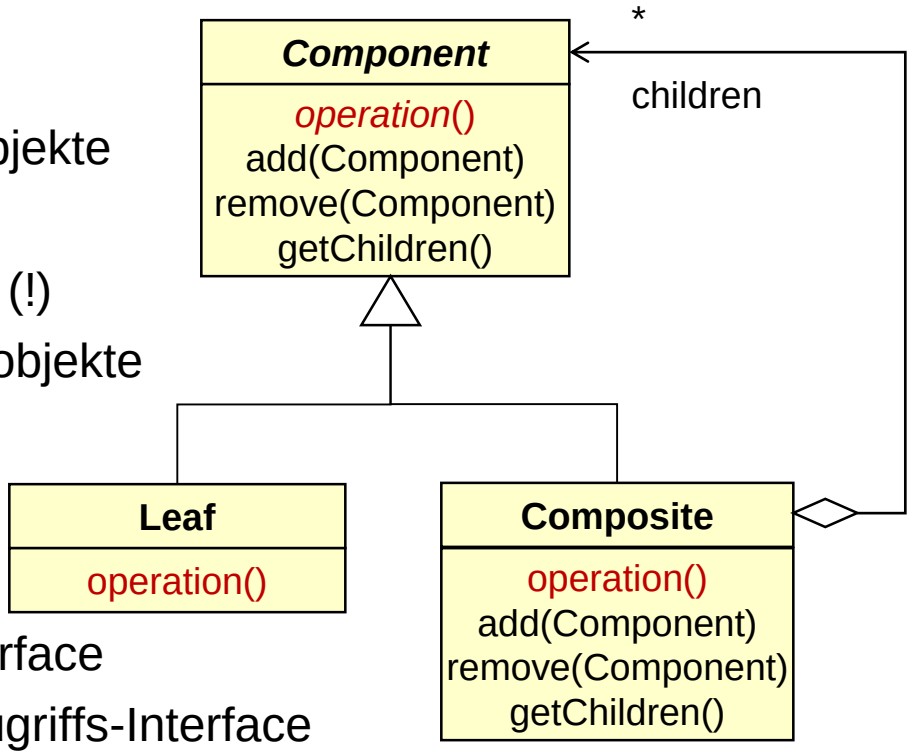


Composite Pattern: Struktur



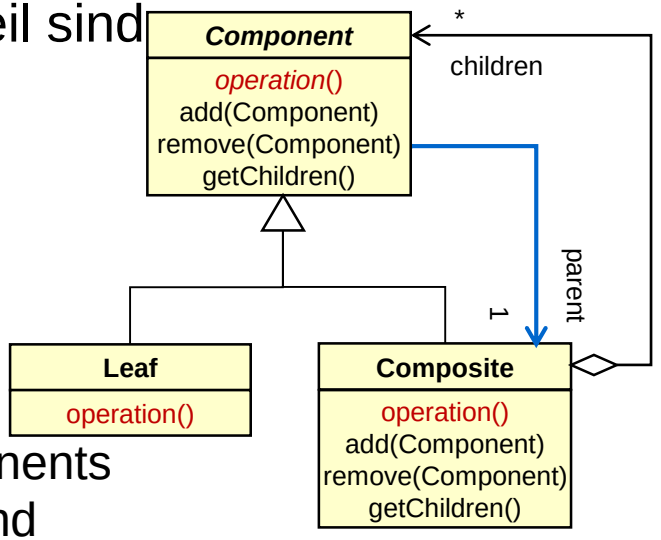
Composite Pattern: Verantwortlichkeiten

- Component (Graphic)
  - gemeinsames Interface aller Teilobjekte
  - default-Verhalten
  - Interface für Zugriff auf Teilobjekte (!)
  - evtl. Interface für Zugriff auf Elternobjekte
- Leaf (Rectangle, Line, Text)
  - "primitive" Teil-Objekte
  - implementieren gemeinsames Interface
  - leere Implementierungen für Teilzugriffs-Interface
- Composite (Picture)
  - speichert Teilobjekte
  - implementiert gemeinsames Interface durch forwarding
  - implementiert Teilzugriffs-Interface



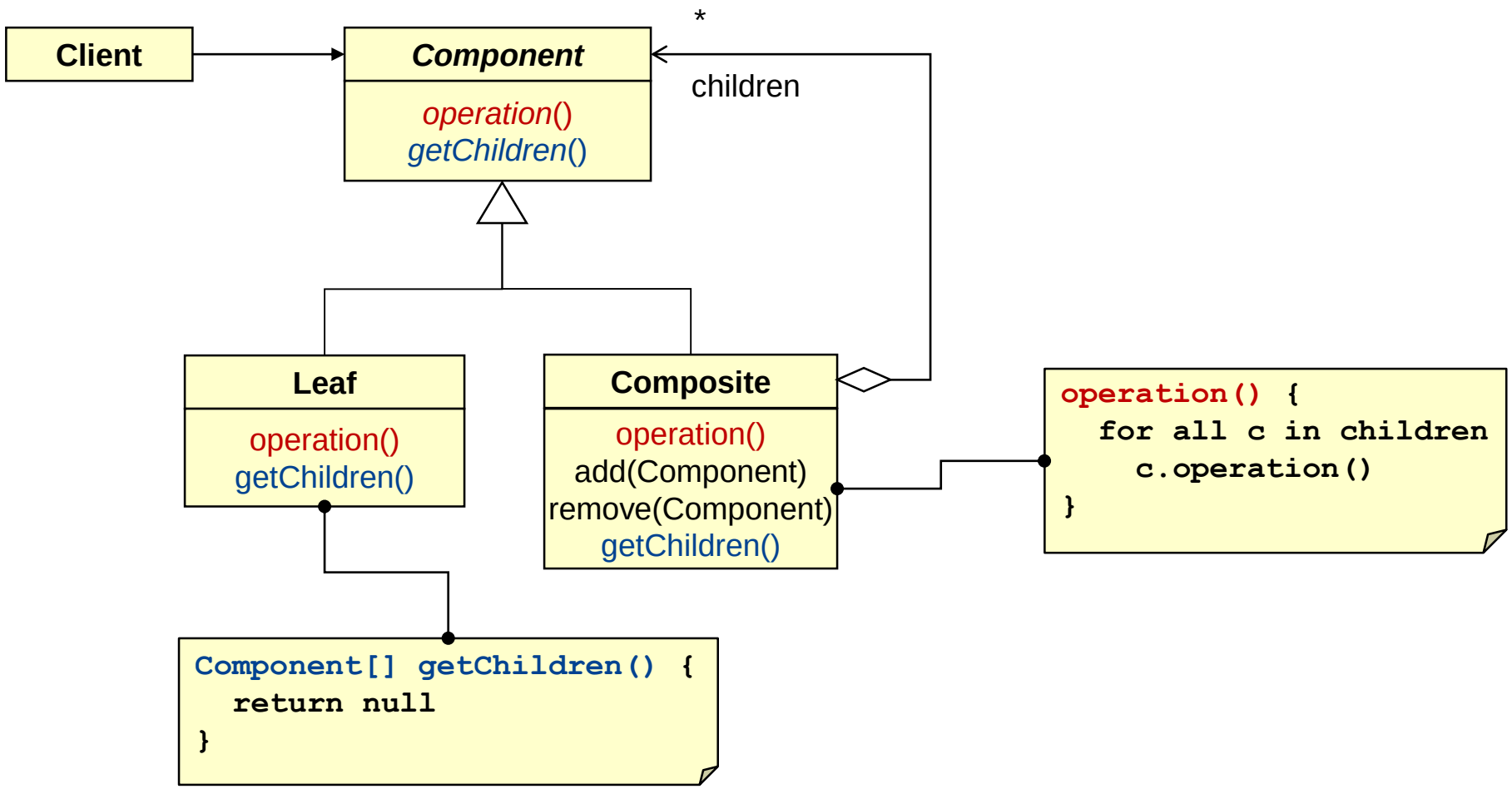
Composite Pattern: Implementierung

- Wenn Composites wissen sollen wovon sie Teil sind
  - Explizite Referenzen auf Composite in Component Klasse
- Wenn mehrere Composites Components gemeinsam nutzen sollen
  - Schließt explizite Referenz der Components auf Composite aus oder erfordert, dass Components wissen, dass sie Teile mehrerer Composites sind



- Component Interface
  - "Sauberes Design": Verwaltungs-Operationen (add, remove) in Composite, da sie für Leafs nicht anwendbar sind.
  - Wunsch nach einheitlicher Behandlung aller Graphic-Objekte durch Clients
    → Verwaltungs-Operationen in Component mit default-Implementierung die nichts tut
    → Leaf-Klassen sind damit zufrieden, Composites müssen die Operationen passend implementieren.

Composite Pattern: Alternative Struktur (add / remove nicht in „Component“)



Composite Pattern: Konsequenzen

- Einheitliche Behandlung
  - Teile
  - Ganzes
- Einfache Clients
  - Dynamisches Binden statt Fallunterscheidungen
- Leichte Erweiterbarkeit
  - neue Leaf-Klassen
  - neue Composite-Klassen
  - ohne Client-Änderung
- Evtl. zu allgemein
  - Einschränkung der Komponenten-Typen schwer
  - „run-time type checks“ (instanceof)

Abgrenzungen

Gemeinsamkeiten und Unterschiede der Pattern

Abgrenzung: Facade, Singleton, Abstract Factory

- Facade
  - Versteckt Subsystem-Details (Typen und Objekte)
- Singleton
  - Facaden sind meist Singletons (man braucht nur eine einzige)
- Abstract Factory
  - Zur Erzeugung konsistenter Sätze von Subsystem-Objekten
  - Als Alternative zu Facade → "Verstecken" plattform-spezifischer Klassen

Abgrenzung: Bridge, Adapter, Abstract Factory

- Bridge
  - antizipierte Entkopplung von Schnittstelle und Implementierung
- Adapter
  - nachträgliche Anpassung der Schnittstelle einer Implementierung
- Abstract Factory
  - nutzbar zur Erzeugung und Konfiguration einer Bridge

Abgrenzung: Factory Method, Abstract Factory, Prototype

- Factory Method
  - Verzögerte Entscheidung über die Klasse eines zu erzeugenden Objektes
  - Entscheidung wird erst in Unterklasse getroffen durch passendes Überschreiben der Factory Method
- Abstract Factory
  - Gruppiert viele Factory Methods die „zusammengehörige“ Objekte erzeugen
  - Erzeugt feste Menge von Objekten von festen Objekttypen
  - Basiert typischerweise auch auf dem Überschreiben von Methoden in Unterklassen, ist somit nicht mehr zur Laufzeit veränderbar
- Prototype
  - Basiert nicht auf Unterklassenbildung sondern auf kopieren von „Musterobjekten“ (Prototypen)
  - Dynamischste Variante, da die zu clonenden Musterobjekte zur Laufzeit ausgetauscht oder verändert werden können

Abgrenzung: Proxy, Adapter, Decorator

- Proxy
  - gleiches Interface
  - kontrolliert Zugriff
  - "Implementierungs-Detail"
- Adapter
  - verschiedene Interfaces
  - ähnlich Protection-Proxy
  - "Implementierungs-Detail"
- Decorator
  - erweitertes Interface
  - zusätzliche Funktionen
  - „konzeptionelle Eigenschaft“

universitätbonn

„Patterns Create Architectures“

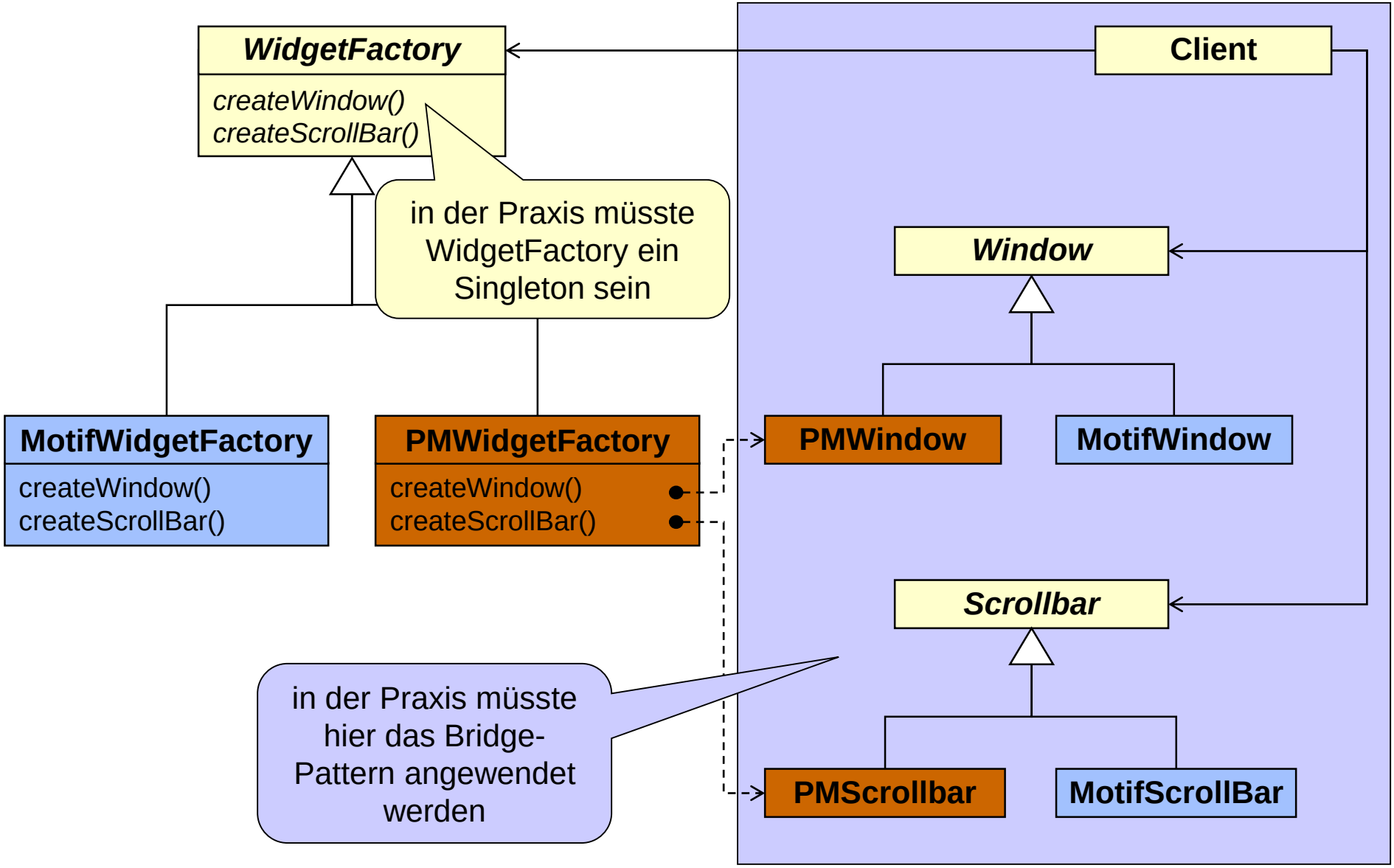
Ein Beispiel zum Zusammenspiel von Patterns

Bridge & Abstract Factory & Singleton

„Patterns Create Architectures“

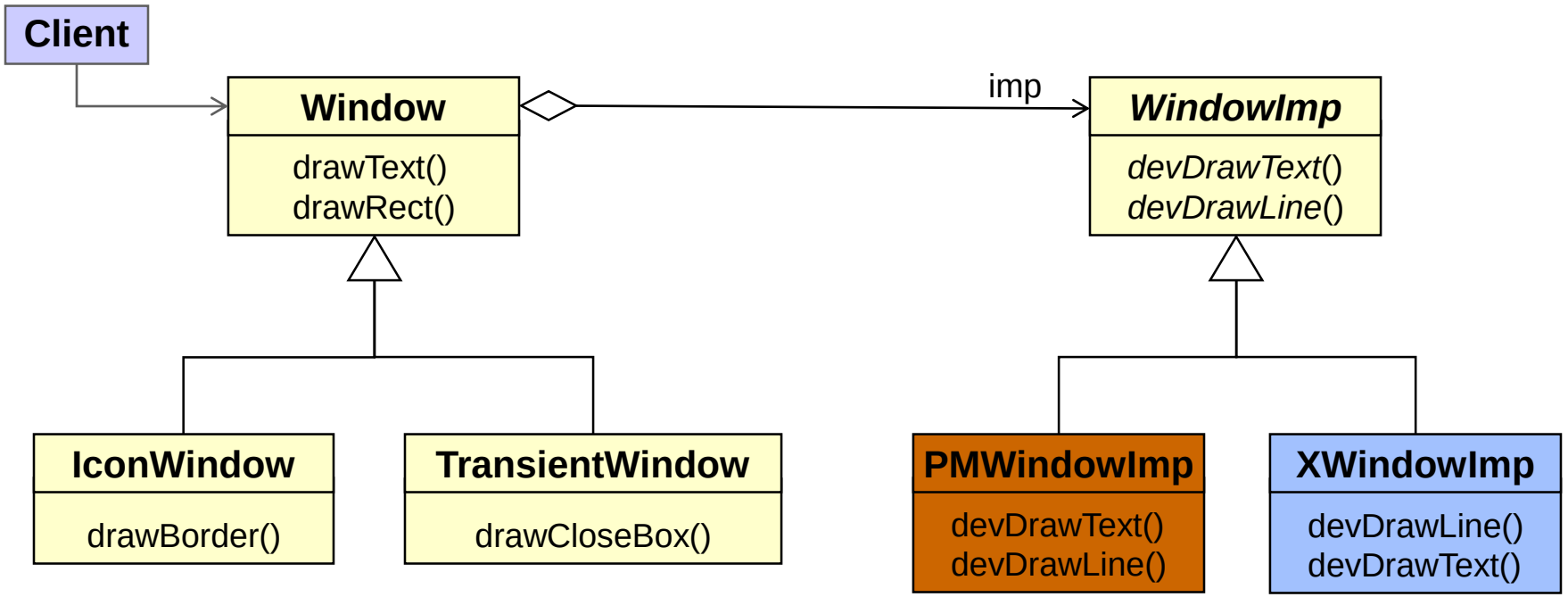
- Idee
  - Wenn man Patterns wohlüberlegt zusammen verwendet, entsteht ein Grossteil einer Software-Architektur aus dem Zusammenspiel der Patterns.
- Beispiel
  - Zusammenspiel von Bridge, Factory und Singleton

Erinnerung ans Abstract Factory Pattern

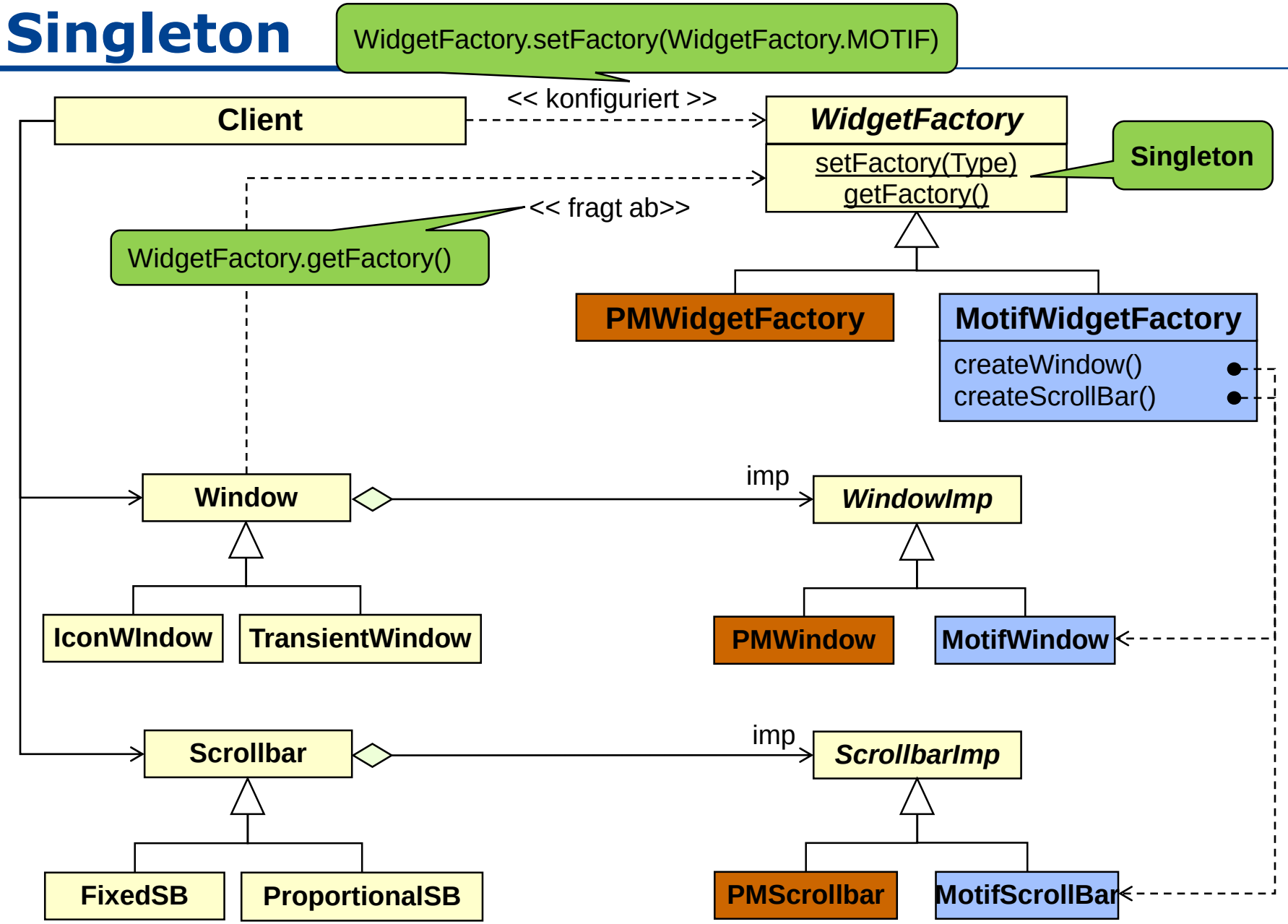


Erinnerung ans Bridge Pattern

- Trennung von
  - Konzeptueller Hierarchie
  - Implementierungs-Hierarchie



Zusammenspiel: Bridge, Factory, Singleton



Rückblick: Was nützen Patterns?

Nutzen von Design Patterns (1)

- Abstraktionen identifizieren, die kein Gegenstück in der realen Welt / dem Analyse-Modell haben
  - Beispiel:
    - ⇨ Command Pattern
    - ⇨ Composite Pattern
    - ⇨ Strategy
    - ⇨ State
  - **"Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.“**
    [Gamma et al. 1995, p. 11]

Nutzen von Design Patterns (2)

- Granularität der Objekte festlegen
  - Command
    - ⇨ einzelne Aktion
  - Visitor
    - ⇨ Gruppe von konsistenten Aktionen
  - Facade
    - ⇨ ein "Riesen-Objekt"
  - Flyweight
    - ⇨ viele kleine, gemeinsam verwendbare Teilobjekte
  - Builder
    - ⇨ Komposition von Gesamt-Objekt aus Teilobjekten

Nutzen von Design Patterns (3)

- Schnittstellen identifizieren
  - was gehört dazu
  - was gehört nicht dazu (Bsp. Memento)

- Beziehungen zwischen Schnittstellen festlegen
  - Subtyping
    - ⇨ Proxy
    - ⇨ Decorator
  - Je eine Methode pro Klasse aus einer anderen Objekthierarchie
    - ⇨ Abstract Factory
    - ⇨ Visitor
    - ⇨ Builder

Nutzen von Design Patterns (4)

- Wahl der Implementierung
  - Interface, Abstrakte Klasse oder konkrete Klasse?
    - ⇨ Grundthema fast aller Patterns
  - Abstrakte Methode oder Hook Methode?
    - ⇨ von template method aufgerufene Methoden
  - Overriding oder fixe Implementierung?
    - ⇨ Factory method
    - ⇨ Template method
  - Vererbung oder Subtyp-Beziehung?
    - ⇨ Adapter, Decorator, State, Strategy, Command, Chain of Responsibility → Gemeinsamer Obertyp, nicht gemeinsame Implementierung
  - Vererbung oder Aggregation?
    - ⇨ Vererbung ist statisch, Aggregation dynamisch
    - ⇨ Wenn das „geerbte“ nicht in der Schnittstelle auftauchen soll: Aggregation und Anfrageweiterleitung nutzen (anstatt Vererbung)
    - ⇨ Siehe alle "split object patterns"

Nutzen von Design Patterns (5):

Patterns verkörpern Grundprinzipien guten Designs

- Implementierungen austauschbar machen
  - Typdeklarationen mit Interfaces statt mit Klassen
  - Design an Interfaces orientieren, nicht an Klassen
  - ➢ Client-Code wiederverwendbar für neue Implementierungen des gleichen Interface
- Objekt-Erzeugung änderbar gestalten
  - "Creational Patterns" statt "new MyClass()"
  - ➢ Client-Code wiederverwendbar für neue Implementierungen des gleichen Interface
- Funktionalität änderbar gestalten
  - Aggregation und Forwarding statt Vererbung
  - ➢ späte Konfigurierbarkeit, Dynamik
  - ➢ weniger implizite Abhängigkeiten (kein "fragile base class problem")

Nichtfunktionale Anforderungen geben Hinweise zur Nutzung von Entwurfsmustern

Identifikation von Entwurfsmustern anhand von Schlüsselwörtern in der Beschreibung nichtfunktionaler Anforderungen

- Analog zu Abbott's Technik bei der Objektidentifikation

- Facade Pattern
  - „stellt Dienst bereit"
  - „muss mit einer *Menge* existierender Objekte zusammenarbeiten",
- Adapter Pattern
  - „muss mit einem existierenden Objekt zusammenarbeiten"
- Bridge Pattern
  - „muss sich um die Schnittstelle zu unterschiedlichen Systemen kümmern von denen einige erst noch entwickelt werden."
  - „ein erster Prototyp muss vorgeführt werden"

Nichtfunktionale Anforderungen geben Hinweise zur Nutzung von Entwurfsmustern (Fortsetzung)

- Proxy Pattern
  - “muss ortstransparent sein”
- Observer Pattern
  - “muss erweiterbar sein”, “muss skalierbar sein”
- Strategy Pattern
  - “muss Funktionalität X in unterschiedlichen, dynamisch auswählbaren Ausprägungen bereitstellen können”
- Composite Pattern
  - „rekursive Struktur“, “komplexe Struktur”
  - “muss variable Tiefe und Breite haben”
- Abstract Factory Pattern
  - “Herstellerunabhängig”,
  - “Geräteunabhängig”,
  - “muss eine Produktfamilie unterstützen”

Pattern ▸ Zusammenfassung

- Betonung von Praktikabilität
  - Patterns sind bekannte Lösungen für erwiesenermaßen wiederkehrende Probleme
  - Lösungen, die sich noch nicht in der Praxis bewährt haben, sind keine Pattern
- Allgemeine Anwendbarkeit
  - Ein Pattern lässt sich in jeder objektorientierten Sprache umsetzen
  - Auf Sprachspezifika (z.B. operator Overloading) basierende „Entwurfsrezepte“ nennt man „Idiome“
- Patterns sind kein Allheilmittel
  - Originalität bei der Anwendung von Patterns ist nach wie vor gefragt.
  - Es muss immer noch abgewogen werden, welche Patterns eingesetzt werden.
- Gesamteffekt
  - Aufgabe des Softwarearchitekten verlagert sich von der Erfindung des Rades zur Auswahl des richtigen Rades und seiner kreativen Anwendung

Literatur

- Erich Gamma, Richard Helm, Ralp Johnson, John Vlissides (Gang of Four): "Design Patterns – Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995

- Frank Buschmann, Regine Meunier, Hans Rohnbert, Peter Sommerlad, Michael Stal (Gang of Five) : "Pattern-Oriented Software Architecture – A System of Patterns" John Wiley & Sons, 1996

- Serge Demeyer, Stephan Ducasse, Oscar Nierstrasz: "Object Oriented Reengineering Patterns", Morgan Kaufman, 2002

- Patterns im WWW
  - Portland Pattern Repository: http://c2.com/ppr
  - Hillside Group Patterns Library: http://www.hillside.net/patterns
  - Brad Appleton: Patterns and Software: Essential Concepts and Terminology http://www.bradapp.net/docs/patterns-intro.html
  - Doug Lea, Patterns-Discussion FAQ, http://gee.oswego.edu/dl/pd-FAQ/pd-FAQ.html
  - Buchliste: http://c2.com/cgi/wiki?PatternRelatedBookList