1 Phasen

- Klassen-Entwurf
  - ➤ Finde Software-Klassen für die Klassen des Analysemodells
  - ➤ Zerteile Analysemodell-Klassen
  - ➤ Entferne unnötige Klassen des Analysemodells
  - ➤ Füge neue Klassen hinzu (z. B. Listen oder Hashtab. zur Verwaltung)
  - ! Objektgrenzen müssen erhalten bleiben!
    Analyse → Design → Implementierung (Traceablity)
- Systementwurf
  - ➤ Nicht-problembezogene Aspekte
    (Verteilung, Nebenläufigkeit, Betriebmittel, Systemschnittstellen, …)
- Programmentwurf
  - ➤ Programmiersprache
  - ➤ Fehlerbehandlung (Exceptions, Rückgabewerte)
  - ➤ Performance-Aspekte

C.10 OOA / OOD - Zusammenfassung

- OOA
  WAS soll mein System tun — nicht WIE
  - ➤ Anforderungsanalyse
  - ➤ Objekte finden und strukturieren
  - ➤ Interaktionen anlysieren
- OOD
  WIE soll mein System arbeiten
  - ➤ Integriere Aspekte der Ausführungsumgebung
  - ➤ treffe strategische Entscheidungen
  - ➤ verfeinere das Objektmodell zu einem implementierbaren Modell

C.11 Entwurfsmuster (Design Patterns)

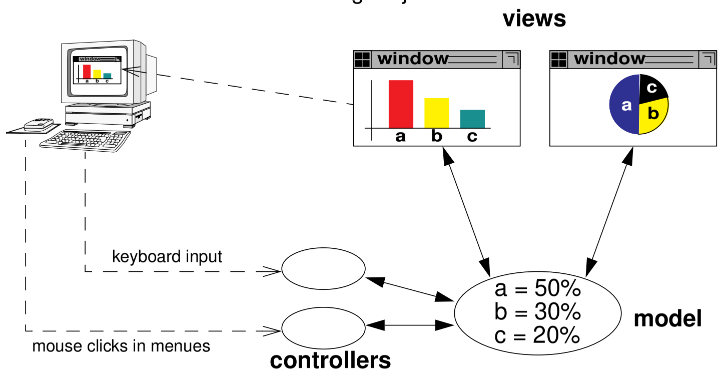
[GHJ+97]

Each pattern describes a problem which occurs over and over again
in our environment,
and then describes the core of the solution to that problem,
in such a way that you can use this solution a million times over,
whithout ever doing it the same way twice
[Christopher Alexander, 1977 —
talking about patterns in buildings and towns]

- Entwurfsmuster sind *Lösungen* für *Probleme*, die auftreten, wenn Software in einem bestimmten *Zusammenhang* entwickelt wird
- Muster erfassen die statische und dynamische *Struktur* und die *Zusammenarbeit* zwischen den wesentlichen *Objekten* in einem Software-Entwurf
- Klassen / Klassenbibliotheken = Wiederverwendung von Implementation
  Entwurfsmuster = Wiederverwendung von Entwürfen

1 Beispiel: Smalltalk’s Model/View/Controller

- Grundlegendes Konzept zum Aufbau von Benutzerschnittstellen
  - *Model*: das Anwendungsobjekt
  - *View*: Bildschirmdarstellung des Anwendungsobjekts
  - *Controller*: nimmt Benutzereingaben entgegen und modfiziert Anwendungsobjekt



Das MVC-Konzept wurde eingeführt, um das eigentliche Anwendungsobjekt von seiner Darstellung am Bildschirm und von der Bearbeitung von Benutzereingaben zur Manipulation zu trennen. Ziel dieser Vorgehensweise ist flexiblere Erweiterbarkeit und Wiederverwendbarkeit.

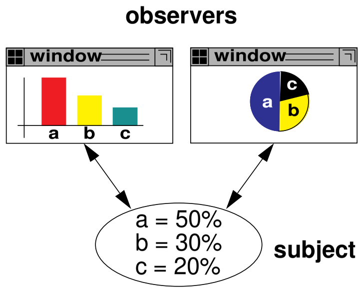
- Model und View werden durch ein subscribe/notify-Protokoll entkoppelt. Ein View meldet sich beim Model an (subscribe) und wird ab diesem Zeitpunkt über Zustandsänderungen durch einen Methodenaufruf (notify) informiert.
  - Auf diese Weise können beliebige Views zur Darstellung eines Objekts eingesetzt werden (prinzipiell auch mehrere gleichzeitig), solange sie die notify-Aufrufe des Models verstehen.
  - Selbst wenn das Model nur das subscribe eines Views unterstützt, können problemlos mehrere Views angekoppelt werden, indem man einfach einen "Verteiler-View" zwischenschaltet, an dem sich dann die eigentlichen Views anmelden.
- Benutzereingaben (Mausklicks, Eingaben über Kontrolltasten, …) werden von einem Controller-Objekt in Methodenaufrufe an dem Model umgesetzt. Verschiedene Benutzer (Anfänger, routiniertes Personal) können unterschiedliche Controller benutzen, ohne dass das eigentliche Anwendungsobjekt hierüber etwas wissen muss. Auch für unterschiedliche Views können jeweils dazu passende Controller eingesetzt werden.

In dem MVC-Konzept kann man verschiedene Design Patterns identifizieren:

1 Beispiel: Smalltalk's Model/View/Controller (2)

Observer pattern

- *View* wird von *Model* entkoppelt
- *View* = *Observer*
- *Model* = Subjekt
- subscribe/notify-Protokoll
- *Observer* unabhängig von Subjekt

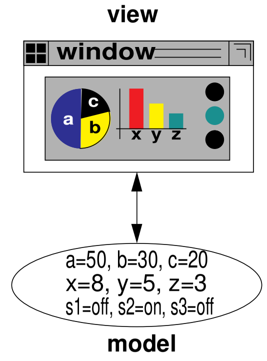


1 Beispiel: Smalltalk's Model/View/Controller (3)

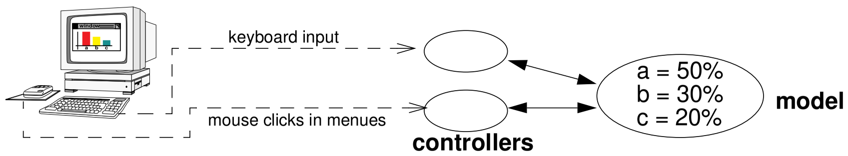
- MVC erlaubt geschachtelte *Views*
  - ein geschachtelter *View* ist selbst ein *View* (`CompositeView` = Unterklasse von `View`)

Composite design pattern

- Klassen-Hierarchie
- primitive Objekte (z.B. Linie, Kreis, Polygon)
- kompatible Verbundobjekte, die einfache Objekte zu komplexeren (z. B. Bild) zusammenfassen
- Verbundobjekte können statt einfacher Objekte verwendet werden



1 Beispiel: Smalltalk's Model/View/Controller (4)



- Benutzereingaben werden von eigenem *Controller*-Objekt angenommen, das Umsetzung in Methodenaufrufe am *Model* vornimmt

➥ **Strategy pattern**

- ➤ *Strategy*-Objekt steht für einen Algorithmus
- ➤ kann statisch oder dynamisch ersetzt werden — unabhängig vom *Model*
- ➤ *Strategy*-Objekte können unterschiedliche Varianten eines Algorithmus kapseln
- ➤ *Strategy*-Objekte können komplexe Datenstrukturen kapseln

2 Elemente eines Entwurfsmusters

- Pattern-Name
- Problem
  - ➤ beschreibt das Problem und den Kontext
- Lösung
  - ➤ beschreibt
    - die Elemente,
    - ihre Beziehung unterienander,
    - Verantwortlichkeiten,
    - das Zusammenwirken
- Resultate
  - ➤ Ergebnisse
  - ➤ Auswirkungen, Nebenwirkungen
    (Probleme in Bezug auf Platz- und Zeitbedarf, Programmiersprache, Implementierung, …)

3 Design Pattern Space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

In der Literatur ist heute eine große Zahl weiterer Entwurfsmuster zu finden - häufig abgestimmt auf bestimmte Anwendungs- oder Problembereiche. Gamma, Helm, Johnson und Vlissides haben sich in ihren Arbeiten bemüht, die Menge der beschriebenen Entwurfsmuster möglichst klein und universell zu halten. Gamma ist der Ansicht, dass ein Großteil der seit dem von anderen publizierten Entwurfsmuster eigentlich auf die ursprünglich identifizierten Muster zurückführbar ist.

Dies bedeutet natürlich nicht, dass die Entwicklung spezieller Entwurfsmuster grundsätzlich nicht sinnvoll ist. Eine gute Kenntnis der "Basis-Muster" und ihrer Anwendung kann einem bei die Bewertung, Einordnung und Auswahl spezieller Entwurfsmuster für spezielle Anwendungsbereiche aber stark erleichtern.

Beispiele:

- Abstract Factory: Objekt mit einer Schnittstelle zum Erzeugen von Objekten
- Builder: Objekt zur Erzeugung eines komplexen Objektaggregats
- Adapter (Wrapper): Passe Schnittstelle an andere, vom Client erwartete Schnittstelle an
- Bridge (Handle/Body): Entkopplung von Abstraktion und Implementierung
- Proxy: Stellvertreterobjekt
- Action: Befehle als Objekt kapseln
- Iterator: Sequentieller Zugriff auf Elemente eines zusammengesetzten Objekts
- Memento: Zustand eines Objekts erfassen (um ihn später rekonstruieren zu können)