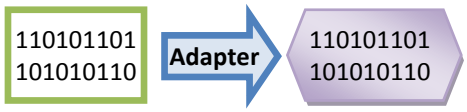
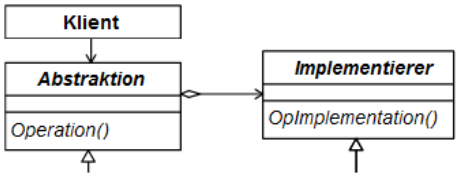
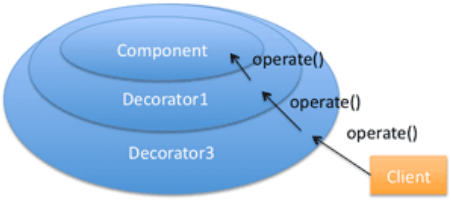
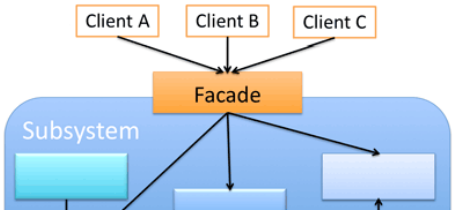
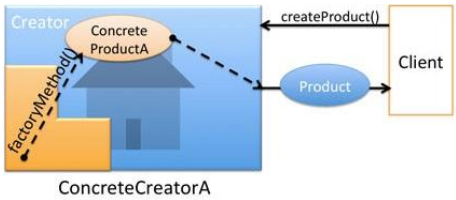
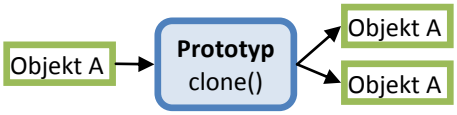
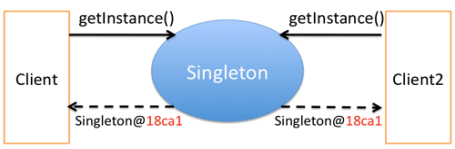
ENTWURFSMUSTER (PATTERNS)

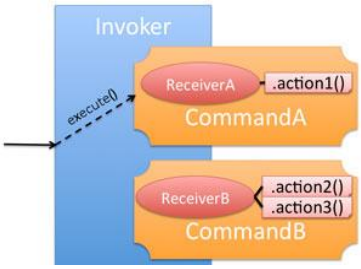
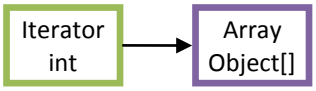
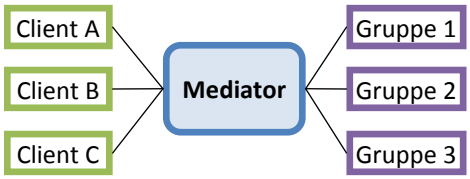
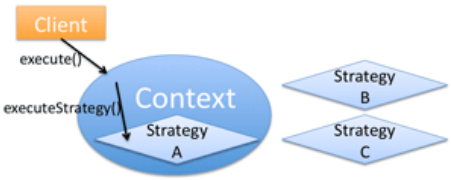
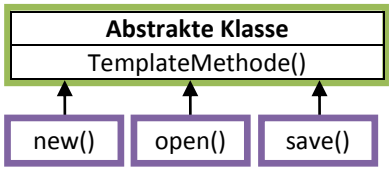
Strukturmuster (Structural Patterns)

Adapter Ähnliche Klassen kompatibel machen		**Konzept** Erstellen einer Adapter-Klasse, die von einer bestehenden Klasse abgeleitet wird. **Verwendung** Bei Kompatibilitätsproblemen (engl/deutsch)
Bridge (Handle) Verbindung zweier Klassenbäume		**Konzept** Klassenbaum in Abstraktion und Implementation aufteilen. Wrapper (Funktionen) in der Abstraktion rufen die Implementation auf. **Verwendung** Mehrere Betriebssysteme + Einfaches pflegen und erweitern; reduzierte Komplexität
Decorator Klasse mit weiteren Funktionen schmücken		**Konzept** Dekorierer wird von der Klasse abgeleitet **Verwendung** Menuleiste mit Effekte ausstatten Text verändern (gross, klein, farbig, …) + bestehende Klasse wird nicht verändert
Facade Delegiert die Funktionalität des Subsystem		**Konzept** Facade kennt die Funktionalität des Subsystems **Verwendung** Bei sehr komplexen Systemen E-Mail, Sekretärin + Vereinfacht den Umgang mit dem Subsystem
Kompositum		kommt nicht
Proxy (Surrogat) Verwalten von Zugriffsrechten		**Konzept** Proxy ist für die Ressourcenplanung verantwortlich **Einteilung** Remote-Proxy - verschlüsselt Anfrage Schutzproxy - erkennt Zugriffsberechtigung Virtuelles-Proxy - ersetz komplexes Objekt durch Einfaches + transparenter Zugriff; optimiertes Laden

Erzeugungsmuster (Creational Patterns)

Factory Erzeugung von unbekannten Objekten		**Konzept** Subklasse entscheidet, was für Objekte erzeugt werden. Aufruf einer abstrakten Methode, anstatt eines direkten Konstuktoraufruf **Verwedung** z.B. Variierte Darstellung der komplexen Zahlen - Erhöhung der Komplexität; Erhöhter Zeitbedarf
Prototyp Neue Instanzen werden aufgrund von Prototypen erzeugt		**Konzept** implementieren von Cloneable; clone()-Methode **Verwendung** Erzeugung weiterer Instanzen teuer ist **Typen** Flache Kopie („Shallow Cloning“) - lässt Verweise stehen Tiefer Kopie („Deep Cloning“) - kopiert auch Referenzen + schnelles, angepasstes Erzeugen (evt. erst bei Laufzeit) - aufwendige Erstellung
Singleton Objekt darf nur einmal existieren		**Konzept** privater Konstruktor, private Variable, globale getInstance()-Methode **Verwendung** Instanzen, die einmal vorkommen dürfen. Printmanager - Sollte zurückhaltend eingesetzt werden.

Verhaltensmuster (Behavioral Patterns)

Chain of Responsibility Anfrage bearbeiten mit Zuständig-keitskette		**Konzept** Verkettung von mehreren Objekten Anfrage gleitet an der Kette entlang, bis sie bearbeitet wird + Klient kennt Aufbau nicht; Glieder kennen nur Nachfolger - Keine Garantie auf Beabeitung = Endlosschleife
Command (Action) Aus Befehlen werden Objekte		**Konzept** Erstellen von Objekten mit Action-Methoden Invoker („Aktionsmanager“) verwaltet diese Objekte **Verwendung** Erhöhte Möglichkeiten mit Befehlen + Möglichkeit zum: verändern, protokollieren, rückgängig machen, in Warteschlange legen, usw.
Iterator (Curser) Zugriff auf unbekannte Datenstruktur		**Konzept** int-Wert zeigt auf den aktuellen Wert im Array + Datenstruktur bleibt verborgen - Erhöhte Laufzeit und Speicherkosten
Mediator Vermittler als zentrale Stelle		**Konzept** Vermittler steuert kooperatives Verhalten **Verwendung** Komplexe Kooperation, unbekannte Kollegen Chatrooms, Tower + weniger Referenzen, Objekte sind unabhängig - komplexer Mediator
Memento (Token) Erinnerung		**Konzept** speichert den internen Status eines Objekts Originator erstellt Memento, wird im Caretaker gespeichert **Verwendung** undo/redo, Protokollierung, Sicherung + Sicherung kann nicht verändert werden - Kosten bei der Aufbewahrung **Unterschiede zum Command-Pattern** Speichert das ganze Objekt, nicht nur die Änderungen
Observer		kommt nicht
State Zustandsautomat		**Konzept** Jeder Zustand hat eine Klasse, Zustandsänderung erzeugt Objekt **Verwendung** Zustandsautomat, Verbindungen, Werkzeuge + Erweiterbarkeit, einfach verständlich, wiederverwenden - hohe Klassenanzahl, hoher Implementierunsaufwand
Strategy (Policy) Austauschbare Algorithmen		**Konzept** Strategy definiert eine Schnittstelle zu den Alg. **Verwendung** viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden; austauschbare Algorithmen + verbergen von Daten innerhalb eines Alg. vor Client
Template Schablone um ein Algorithmus variabel zu halten		**Konzept** Abstrakte Klasse definiert das Skelett des Algorith. Einschubmethoden ermöglichen eine konkrete Ausformung **Verwendung** Vermeidung von doppeltem Code Erweiterung durch Unterklassen zu kontrollieren + gut modifizierbarer Algorithmus
Visitor		kommt nicht

Quelle Bilder: http://www.philipphauer.de/study/se/design-pattern