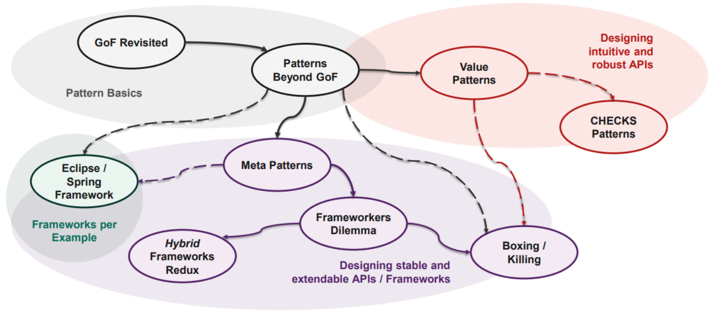
Patterns & Frameworks - HS2021

joel.schaltegger@ost.ch

1 Introduction



1.1 Pattern Definition

- Descriptions of successful engineering stories with a well-known intent
  - A ddress recurring problem
  - D escripe generic solution that worked
- Tell about the forces of the problem (why is the problem hard)
- Tell about the engineering trade-offs to take (Benefits / Liabilities)
- Solution (Implementation)

1.2 Type of Patterns

- Architecture Patterns (Waiting Room)
- Software Patterns
  - D esign Pattern (Elements of Reusable Object-Oriented Sofware → GoF)
  - P attern-oriented Software Architecture (POSA)
- Organizational Patterns
- Learning and Teaching Patterns
- Documentation Patterns

1.3 Pattern Formats

1.3.1 POSA

- Name
- Intent
- Problem
- Solution
- Benefits/Liabilities

1.3.2 Fault Tolerance

- Name
- Intent
- Solution
- Benefits/Liabilities

1.3.3 MAPI

- Name
- Intent
- Consequences (Benefits/Liabilities)

1.3.4 Game Programming Patterns

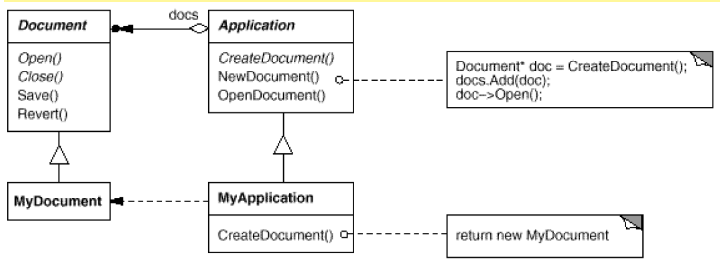
- Name
- Problem
- Engineering Story that worked
- Benefits/Liabilities
- Solution

1.4 What are Patterns not?

- Silver bullet
- Novices Tool
- Ready Made Components
- Means to turn off your brain

2 GoF Patterns

2.1 Factory Method (Creational)



2.1.1 Beschreibung

Define an interface for creating an object, but let the subclass decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

2.1.2 Vorteile

- Verhindert eine Kopplung zwischen dem Creator und dem konkretem produkt
- Single Responsibility Principle: Der Product Creation Code kann einfach herumgeschoben werden → Macht den Support einfacher
- Open/Closed Principle: Es können neue Produkt-Typen hinzugefügt werden, ohne den Client Code zu zerstören

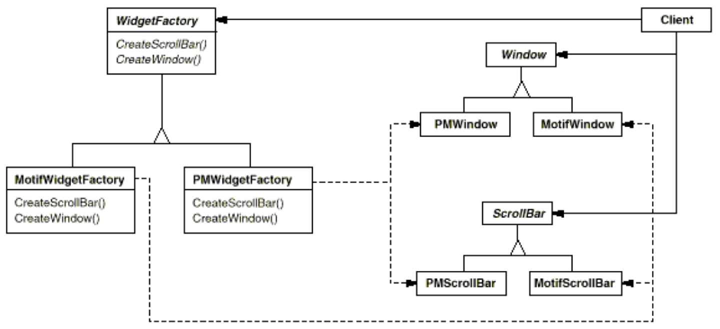
2.1.3 Nachteile

- Code kann komplizierter werden, weil viele Sub-Klassen erstellt werden müssen

2.1.4 Eigene Notizen

Intent: return new MyDocument **Identifikation:** Haben eine Creation-Methode, die Objekte von konkreten Klassen erstellen, diese aber als Objekte von abstrakten Klassen oder Interfaces zurückgibt.

2.2 Abstract Factory (Creational)



2.2.1 Beschreibung

Provide an interface for creating families of related or dependant objects without specifying their concrete classes.

2.2.2 Vorteile

- Man kann sicher sein, dass die Produkte einer Factory miteinander kompatibel sind
- Vermeidet enge Kopplung zwischen konkreten Produkten und Client Code
- Single Responsibility Principle: Der Product Creation Code kann einfach herumgeschoben werden → Macht den Support einfacher
- Open/Close Principle: Es können neue Varianten hinzugefügt werden, ohne existierenden Code zu zerstören

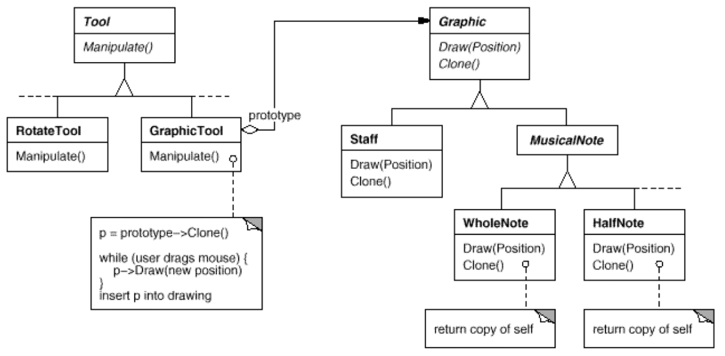
2.2.3 Nachteile

- Der Code kann komplizierter werden, als er sein sollte, da mit diesem Pattern viele neue Interfaces und Klassen erstellt werden

2.2.4 Eigene Notizen

Identifikation: Methoden, die ein Factory-Objekt zurückgeben, welches dann verwendet wird um spezifische Sub-Komponenten zu erstellen
Factory Method: Verwendet eigentlich immer das Pattern Factory Method (grössere Variante davon)
Beispiel 1: Unterschiedliche Buttons für Windows und MacOS. Dabei sind Windows & MacOS die factories
Beispiel 2: Window Manager in einem Linux System (je nachdem, ob Gnome oder KDE installiert ist)

2.3 Prototype (Creational)



2.3.1 Beschreibung

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

2.3.2 Vorteile

- Objekte können geklont werden, ohne Kompplung zu dessen konkreter Klasse
- Wiederholte Initialisierung kann durch das Klonen vermiedenw erden
- Komplexe Objekte können bequemer erstellt werden
- Alternative von Vererbung beim Umgang mit Voreinstellungen für komplexe Objekte

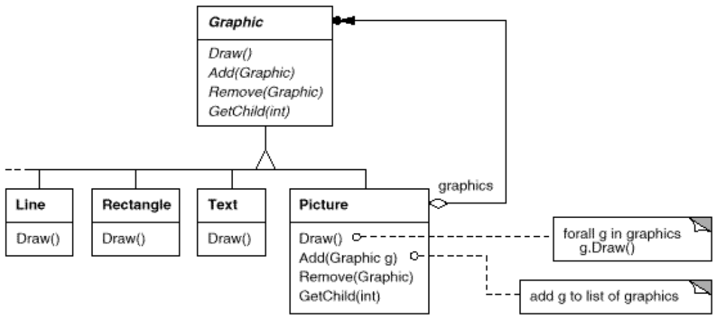
2.3.3 Nachteile

- Sehr tricky Objekte mit zirkularen Referenzen zu klonen

2.3.4 Eigene Notizen

Identifikation: Hat eine clone(), oder copy()-Methode
Beispiel: Musik-Noten darstellen: Die existierenden Noten in einen Manager (Pool) laden und diese jeweils bei Gebrauch klonen
Einsetzen: Ist eine möglichkeit, wenn Daten von externen Quellen sehr ineffizient geladen werden (Clone erstellen um effizienz zu steigern)

2.4 Composite (Structural)



2.4.1 Beschreibung

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

2.4.2 Vorteile

- Bequemer um mit komplexen Baumstrukturen zu arbeiten. Verwende Polymoprhismus und Rekursion zum Vorteil
- Open/Closed Principle Es können neue Element-Typen hinzugefügt werden, ohne den existierenden Code zu zerstören

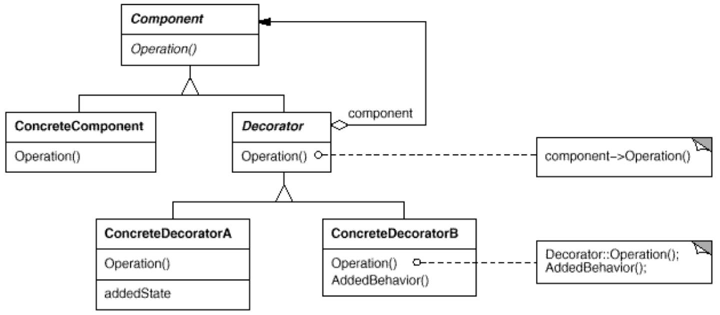
2.4.3 Nachteile

- Es kann schwierig sein, eine gemeinsame Schnittstelle anzubieten

2.4.4 Eigene Notizen

Identifkation: Haben Baum-Struktur und Klassen die Arbeit zu dessen Kinder delegieren

2.5 Decorator (Structural)



2.5.1 Beschreibung

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

2.5.2 Vorteile

- Das Verhalten eines Objekts kann erweitert werden, ohne Sub-Klassen zu erstellen
- Verantwortungen können zur Laufzeit einem Objekt hinzugefügt und entfernt werden
- Mehrere Verhalten können kombiniert werden, indem ein Objekt in mehrere Decorator gewrappt wird
- Single Responsibility Principle: Ein Monolith kann in mehrere kleine Klassen unterteilt werden

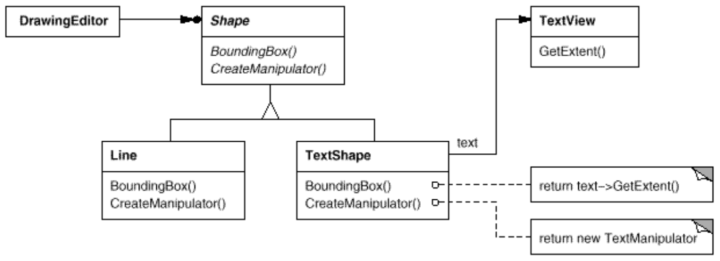
2.5.3 Nachteile

- Es ist schwer ein Wrapper vom Wrapper-Stack zu entfernen
- Es ist schwer ein Decorator so zu implementieren, dass die Reihenfolge keine Rolle spielt
- Der initiale Configuration Code von Layers kann ugly aussehen

2.5.4 Eigene Notizen

Identifikation: Hat Creation-Methode oder Konstruktur die Objekte der selben Klasse akzeptieren, wie die eigene Klasse
Super: Es wird viel mit dem super-Keyword gearbeitet

2.6 Adapter (Structural)



2.6.1 Beschreibung

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

2.6.2 Vorteile

- Single Responsibility Principle: Das Interface oder Data Conversion Code kann von der Business Logik separiert werden
- Open/Closed Principle Es können neue Adapters hinzugefügt werden, ohne den Client Code zu zerstören

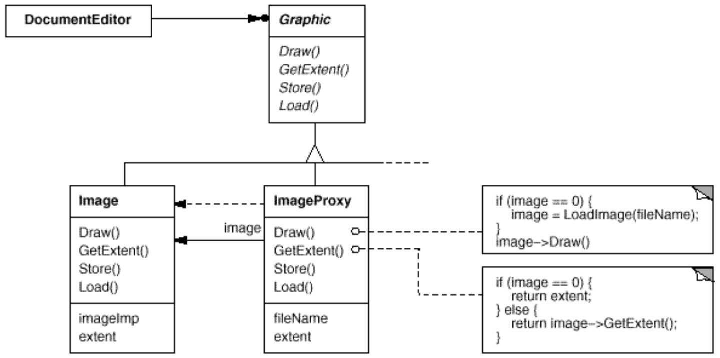
2.6.3 Nachteile

- Die Gesamt-Komplexität des Codes steigt, weil neue Interfaces und Klassen erstellt werden müssen

2.6.4 Eigene Notizen

Identifikation: Nehmen im Konstruktur eine Instanz eines anderen Types. Diese wird verwendet, um in Methoden die Methoden der Instanz auszuführen.
Anwendung: Erbe von dem was du sein möchtest und übernimm im Konstruktor, dass Objekt welches geändert werden soll

2.7 Proxy (Structural)



2.7.1 Beschreibung

Provide a surrogate or placeholder for another object to control access to it.
Bietet einen Stellvertreter oder Platzhalter für ein anderes Objekt. Ein Proxy kontrolliert den Zugrff auf das originial Objekt, was erlaubt vor oder nach dem eigentlichen Request, Code auszuführen.

2.7.2 Vorteile

- Das Service Objekt kann kontrolliert werden, ohne dass der Client etwas davon weiss
- Ermöglicht das Managen des Life-Cycles des Services, wenn sich Clients nicht darum kümmern
- Proxy funktioniert auch, wenn der Service nicht ready oder nicht verfügbar ist
- Open/Closed Principle: Neue Proxies können erstellt werden, ohne den Service oder Client zu verändern

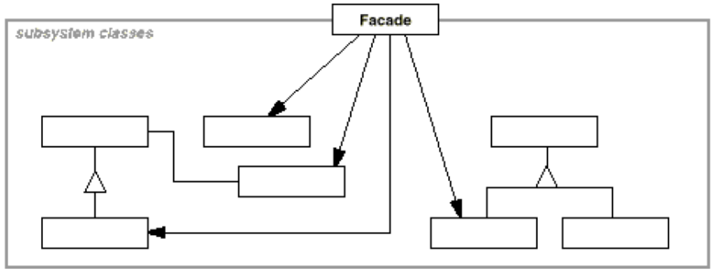
2.7.3 Nachteile

- Code kann komplizierter werden, da neue Klassen erstellt werden müssen
- Antwort des Services kann verzögert werden

2.7.4 Eigene Notizen

Identifikation: Delegieren die eigentliche Arbeit an andere Objekte. Jede Proxy-Methode soll zu einem Service-Objekt referenzieren
Anwendung: Wird sehr oft verwendet, wenn zusätzlicher Code ausgeführt werden soll

2.8 Facade (Structural)



2.8.1 Problem

2.8.2 Beschreibung

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Subsysteme können Libraries, Frameworks oder ein komplexes Set von Klassen sein.

2.8.3 Vorteile

Code kann von der Komplexität des Subsystems isoliert werden

2.8.4 Nachteile

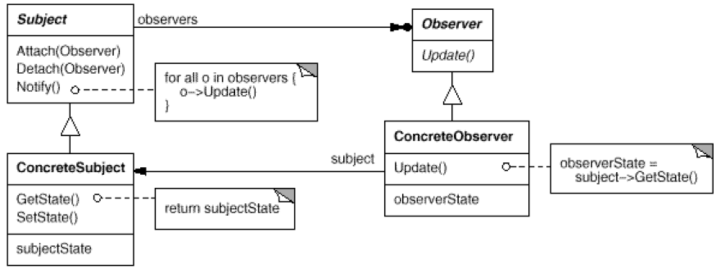
- Facade kann ein God-Object, welches an alle Klassen einer App gekoppelt ist werden

2.8.5 Eigene Notizen

Identifikation: Ist meistens eine Klasse mit einer einfachen Schnittstelle, welche aber alle Arbeit an andere Klassen weiter delegiert. Facade managen oftmals den Life-Cycle der Objekte die sie verwendet.

Auftreten: Passiert sehr viel innerhalb einer einfachen Schnittstelle

2.9 Observer (Behavioral)



2.9.1 Problem

Anstelle, dass Objekt A beim Objekt B regelmässig abfragt, ob es sich geändert hat, benachrichtigt B A, dass es sich geändert hat. Benachrichtige mich wenn.. → Subject informiert alle Observer

2.9.2 Beschreibung

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

2.9.3 Vorteile

- Open/Closed Prinzip: Es können neue Subscriber Klassen erstellt werden, ohne den Publisher ändern zu müssen
- Es können zur Laufzeit beziehungen zwischen Objekten erstellt werden

2.9.4 Nachteile

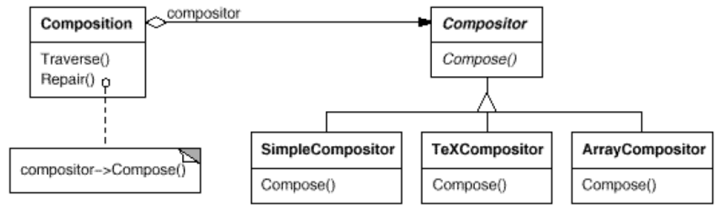
- Subscribers werden in zufälliger Reihenfolge benachrichtigt

2.9.5 Eigene Notizen

Identifikation: Hat Subscription Methoden, die Objekte in einer Liste speichert und eine Update Methode für diese Objekte ausführt.

Beispiel: Click Event Handlers im Web

2.10 Strategy (Behavioral)



2.10.1 Problem

Verschiedene Compositors (Algorithmen) anbieten. Das Objekt Composition verwendet dann denn benötigten Compositor.

2.10.2 Beschreibung

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

2.10.3 Vorteile

- Algorithmen können zur Laufzeit ausgetauscht werden (Polymorphie)
- Isolation der Implementations Details (Algorithmen) gegenüber dem Code, welcher sie verwendet
- Vererbung kann mit Composition ausgetauscht werden
- Open/Close Principle: Es können neue Strategien hinzugefügt werden, ohne den Kontext zu ändern

2.10.4 Nachteile

- Überkompliziert, wenn nur wenige Algorithmen existieren und diese selten ausgetauscht werden
- Clients müssen die unterschiede der Algorithmen kennen, damit der richtige ausgewählt werden kann

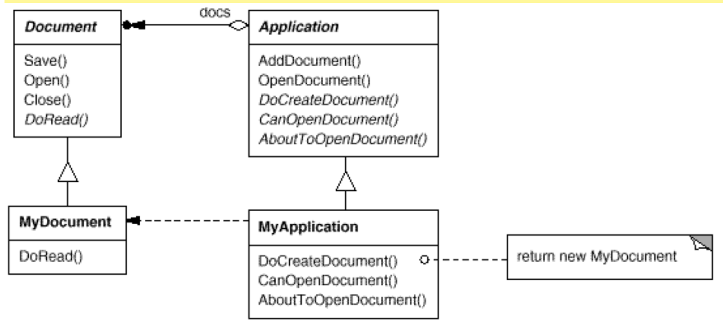
2.10.5 Eigene Notizen

Umsetzung mit: Factory Method, oder beim Erstellen dem Konstruktor übergeben.

Identifikation: Hat eine Methode, die nested Objekte die Arbeit machen lässt und einen Setter, damit das Objekt ausgetauscht werden kann.

Beispiel: Unterschiedliche Algorithmen für einen Routen planer.

2.11 Template Method (Behavioral)



2.11.1 Problem

Austauschen einzelner Schritte eines Workflows.

2.11.2 Beschreibung

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

2.11.3 Vorteile

- Clients können Teile eines Algorithmus austauschen
- Duplizierter Code kann in Super-Klasse verschoben werden

2.11.4 Nachteile

- Klassen können durch das bereitgestellte Gerüst limitiert werden
- Das Likov Substitution Principle kann verletzt werden, indem keine Default-Implementierung möglich ist
- Template Methoden werden schwerer unterhalten, je mehr schritte existieren

2.11.5 Eigene Notizen

Identifikation: Haben Verhaltens-Methoden, die ein "default"-Verhalten in der Basis-Klasse implementiert haben

Application: Die Funktionen der Application können bereits das Patterns Factory Method implementieren. z.B. Beinhaltet die Funktion AddDocument() die Abstrakte Klasse DoCreateDocument()

2.12 Mediator (Vermittler) (Behavioral)

2.12.1 Beschreibung

Reduziert chaotische Abhängigkeiten zwischen Objekten. Das Pattern verhindert direkte Kommunikation zwischen den Objekten und zwingt diese mit dem Madiator-Objekt zusammenzuarbeiten.

2.12.2 Problem

- Object Structures may result in many connections between objects
- In the worst case, every object ends up knowing about every other

Intent:

- How can strong coupling between multiple objects be avoided and communication simplified?

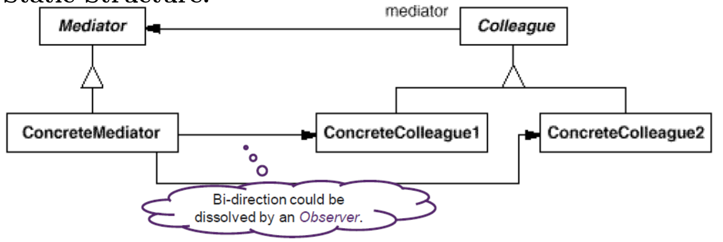
2.12.3 Solution

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and lets you vary their interaction independently.

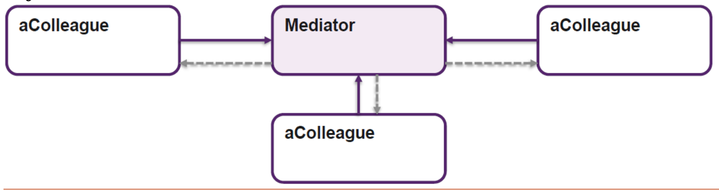
Mediator: Encapsulates how a set of objects interact

Colleaues: Refer to Mediator; this promotes loose coupling

Static Structure:



Dynamics:



2.12.4 Implementation

- Mediator as an Observer
- Colleagues act as Subject

Known Uses:

- Message Bus Systems
- Redux Dispatcher

2.12.5 Vorteile

- Colleague classes may become more reusable, low coupling
- Centralizes control of communication between objects
- Encapsulates protocols
- Single Responsibility Principle: Kommunikation zwischen verschiedenen Komponenten kann zu einem Ort extrahiert werden. Macht es einfacher diese zu Unterhalten
- Open/Closed Principle Es können neue Mediators hinzugefügt werden, ohne die Komponenten zu ändern
- Reduziert die Kopplung zwischen den Komponenten
- Individuelle Komponenten können einfacher wiederverwenet werden

2.12.6 Nachteile:

- Adds complexity
- Single point of failure
- Limits subclassing (of mediator class)
- May result in hard maintainable monoliths
- Mediator kann zu einer God-Klasse werden

2.12.7 Eigene Notizen

Der Controller-Teil von MVC ist ein Synonym für Mediator **Teilnehmer:** Mediator, Colleague

2.13 Memento (Behavioral)

2.13.1 Beschreibung

Erlaubt das Speichern und Wiederherstellen von vorherigen Zuständen eines Objekts, ohne die Implementierungs Details zu enthüllen

2.13.2 Problem

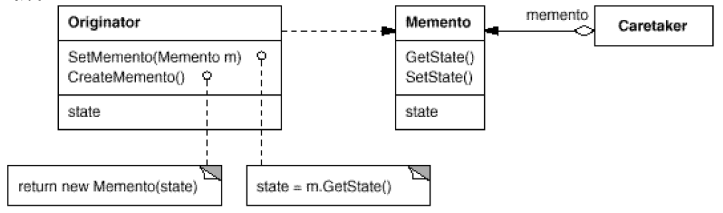
- Sometimes it's necessary to record the internal state of an object
- Objects normally encapsulate their state, making it inaccessible

Intent:

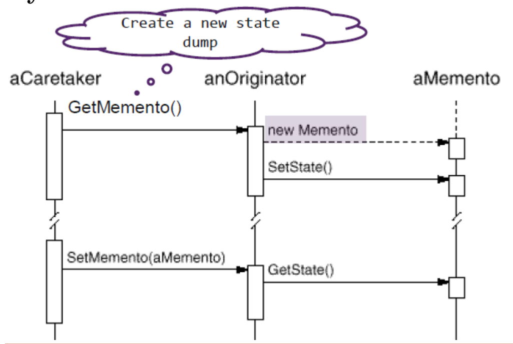
- How can the state of an object be externalized without violating its encapsulation?

2.13.3 Solution

Without violating encapsulation, capture and externalize an objects internal state so that the object can be restored to this state later.



Dynamics:



2.13.4 Participants

Memento

- Stores some/all the internal state of the Originator
- Allows only the originator to access its internal information

Originator

- Can create Memento objects to store its internal state at strategic points
- Can restore own state to what the Memento object dictates

Caretaker

- Stores the Memento objects
- Cannot explore/operate the contents

2.13.5 Implementation

- Originator creates memento and passes over its internal state
- Can be combined with Factory Method
- Declare Originator as *friend* of Memento, so Originator can read out its properties

2.13.6 Vorteile

- Internal State of an object (snapshots) can be saved and restored at any time
- Encapsulation of attributes is not harmed
- State of objects can be restored later
- Originator Code kann vereinfacht werden, indem der Caretaker Rücksicht auf die History des Originator Zustandes nimmt

2.13.7 Nachteile

- Creates a complete copy of the object every time, no diffs (memory usage)
- No direct access to saved state, it must be restored first
- App kann sehr viel RAM brauchen
- Caretakers müssen den Life-Cycle des Originator verfolgen, damit überflüssige Mementos gelöscht werden können

2.13.8 Eigene Notizen

- Ohne Memento müsten die Felder einer Klasse public sein
- Moderne Alternativen sind Serialisierung (Java/C#). Sei es Serialisierung für eine Datenbank, oder ein File-System. Ähnlich zu Future-Token
- Wird in Java oftmals mit Serializable gemacht
- Teilnehmer: Memento, Originator, Caretaker

2.14 Command (Behavioral)

2.14.1 Beschreibung

Wandelt einen Request in ein stand-alone Objekt um, welches alle Informationen zum Request enthaltet. Das erlaubt, dass Requests als Methoden-Argumente verwendet werden, die Request verzögert werden, in einer Warteschlange stehen und erlaubt undoable operationen.

2.14.2 Problem

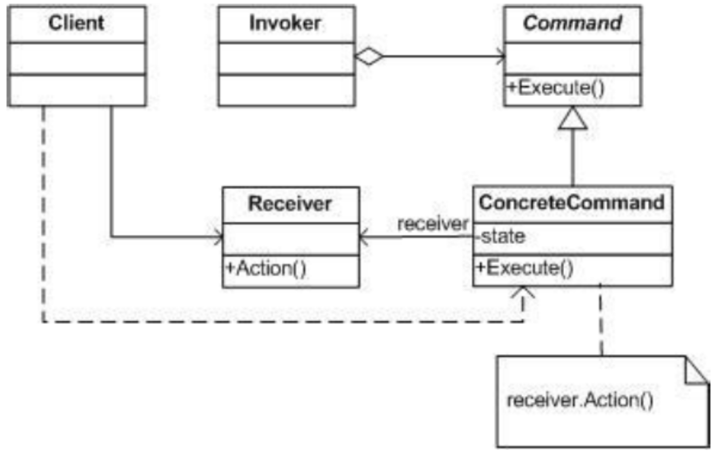
- Decouple the decision of what to execute from the decision of when to execute
- The execution needs an additional parametrization context

Intent:

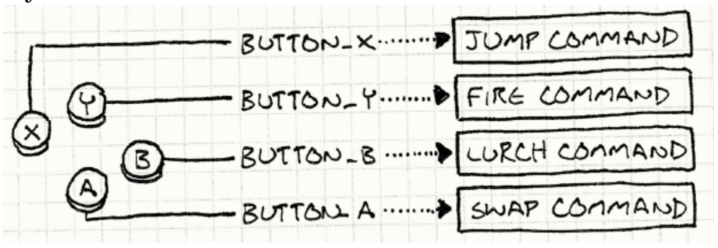
- How can commands be encapsulated, so that they can be parameterized, scheduled, logged and/or undone?

2.14.3 Solution

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operation.



Dynamics:



2.14.4 Vorteile

- The same command can be activated from different objects
- New commands can be introduced quickly and easily
- Command objects can be saved in a command history
- Provides inversion of control, encourages decoupling in both time and space
- Single Responsibility Principle: Klassen die Operationen aufrufen, können von den Klassen die Operationen ausführen, entkoppelt werden
- Open/Closed Principle: Neu Commandos können hinzugefügt werden ohne existierenden Client Code zu zerstören
- Undo/Redo kann implementiert werden
- Das Ausführen von Operationen kann aufgeschoben werden
- Ein Set von einfachen Commands kann zu einem komplexen zusammengestellt werden

2.14.5 Nachteile

- Large designs with many commands can introduce many small command classes mauling the design
- Der Code kann komplizierter werden, weil ein neues Layer zwischen Sender und Empfänger hinzugefügt wird

2.14.6 Eigene Notizen

Identfikation: Wenn es ein Set von ähnlichen Klassen hat, welche spezifische Aktionen darstellen (Copy, Cut, Send, Print etc.). Diese Klassen sollten das selbe Interface implementieren. Zusätzlich hat es meistens eine Klasse, welche diese Aktionen als Argument akzeptiert

- Do und Undo werden im Command gespeichert
- Wird z.B. verwendet für callbacks, Queueing Tasks, Tracking Operations History etc.

2.15 Command Processor (Behavioral)

2.15.1 Beschreibung

Separiere den Request für einen Service von dessen Ausführung. Command Processor Komponente verwaltet Requests als separate Objekte, plant zeitlich dessen Ausführung und bietet zusätzliche Services, wie das Speichern von Request-Objekten für späters undo.

2.15.2 Problem

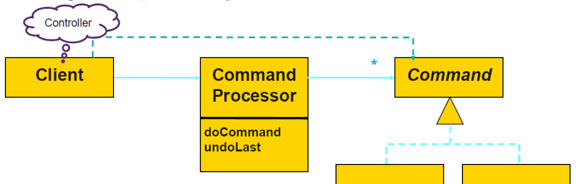
- Common UI applications support do and multiple undo steps
- Steps forward and backward are accessible in a history

Intent:

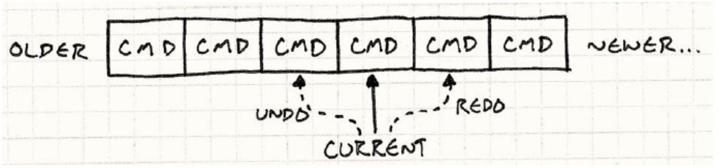
- How could we manage command objects, so the execution is seperated from the request and the execution can be undone later?

2.15.3 Solution

Separate the request for a service from its execution. A command processor component manages requests as separate objects, schedules their execution, and provides additional services such as the storing of request objects for later undo.



Dynamics:



2.15.4 Participants

Command Processor

- A Separate processor object can handle the responsibility for multiple Command objects

Command

- A uniform interface to execute functions

Controller

- Translates requests into commands and transfers commands to Command Processor.

2.15.5 Implementation

- Command Processor contains a *Stack* which holds the command history
- Controller creates the Commands and passes them over to Command Processor
- Creation of Commands may be delegated to a *Simple Factory*

2.15.6 Vorteile

- Flexibility → Command Processor und Controller werden unabhängig von Commands implementiert
- Allows addition of services related to command execution
- Enhances testability → Command Processor kann verwendet werden, um Regression Tests durchzuführen

2.15.7 Nachteile

- Efficiency loss due additional indirection

2.15.8 Eigene Notizen

- Dank dem Command Processor Pattern kann z.B. ein Logger eingeführt werden, dies ist nur mit dem Command nicht möglich! Command ist nur ein Interface!
- Teilnehmer: Command Processor, Command, Controller

2.16 Visitor (Behavioral)

2.16.1 Beschreibung

Erlaubt das Ausführen einer Operation auf den Elementen einer Objekt Struktur. Ermöglicht den Algorithmus vom Objekt zu separieren, auf dem es ausgeführt wird.

2.16.2 Problem

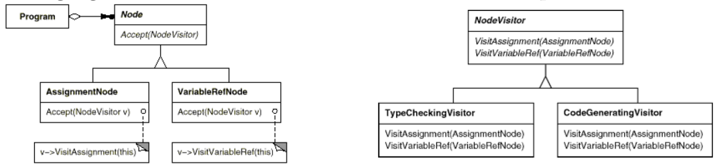
- Operations on specific classes needs to be changed/added without needing to modify these classes
- Different algorithms needed to process an object tree

Intent:

- How can the behaviour on individual elements of a data structure be changed/replaced whout changing the elements?

2.16.3 Solution

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.


Program
Node
Accept(NodeVisitor)
AssignmentNode
Accept(NodeVisitor v)
VariableRefNode
Accept(NodeVisitor v)
v->VisitAssignment(this)
v->VisitVariableRef(this)
NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)

2.16.4 Implementation

- 2 Class Hierarchies (Elements / Visitors)
- Visitors iterate though object hierarchy
- Solves Double-Dispatch problem of single dispatched programming languages

Patterns that combine naturally with Vistor:

- Composite
- Interpreter
- Chain of Responsibility

2.16.5 Vorteile

- Visitor makes adding new operatios easy
- Separates related operations from unrelated ones
- Open/Closed Principle: Neues Verhalten, welches auf Objekten unterschiedlicher Klassen ausgeführt werden kann, ohne die Klassen zu ändern
- Single Responsibility Principle: Mehrere Versionen des selben Verhaltens können in die selbe Klasse verschoben werden
- Es können komplexe Datenstrukturen traversiert werden

2.16.6 Nachteile

- Adding new node classes is hard
- Visiting sequence is fix defined within nodes
- Visitor breaks logic apart
- Alle Visitors müssen geupdatet werden, wenn eine Klasse hinzugefügt oder entfernt wird
- Visitors können nicht auf private Felder zugreifen
- Schlecht wenn die Klassen-Hierarchie sich ändert

2.16.7 Eigene Notizen

- Wenn unterschiedliche Algorithmen verwendet werden, um ein Object-Tree zu verarbeiten
- Visitor werden sehr oft mit Composite, Interpreter oder Chain of Responsibility kombiniert

3 Beyond GoF

3.1 External vs. Internal Iterator

Wenn der Programmierer die Iteration kontrolliert, ist es **External**, sonst **Internal**.

3.2 External Iterator

3.2.1 Problem

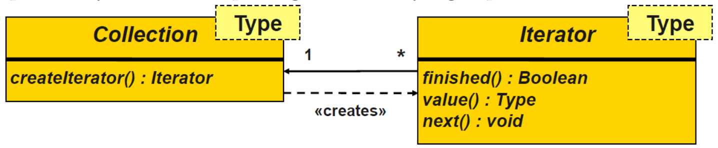
- Iteration through a collection depends on the target implementation
- Separate logic of iteration into an object to allow multiple iteration strategies

Intent:

- How can strong coupling between iteration and collection be avoided, generalized and provided in a collection-optimized manner?

3.2.2 Solution

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



Elementary operations of an Iterator's behaviour:

- Initializing an iteration *new ArrayList().iterator();*
- Checking a completion condition *it.hasNext();*
- Accessing a current target value *var x = it.next();*
- Moving to the next target value *it.next();*

3.2.3 Vorteile

- Provides a single interface to loop though any kind of collection

3.2.4 Nachteile

- Multiple iterators may loop through a collection at the same time → Robustheit ist schwer zu erreichen, wenn sich die Collection ändert
- Life-Cycle Management of iterator objects → Braucht vlt Disposal-Methode oder Iterator muss die Collection überwachen
- Close coupling between Iterator and Collection class
- Indexing might be more intuitive for programmers

3.2.5 Eigene Notizen

- for(Class : collection) → Ist External
- C-Like Iterationen

3.3 Enumeration Method (Internal Iterator)

3.3.1 Problem:

- Iteration management is performed by the collection's user
- Avoid state management between collection and iteration

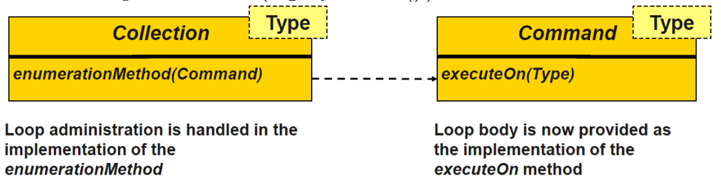
Intent:

- How can a collection be iterated considering the collection state and furthermore state management be reduced?

3.3.2 Solution

Support encapsulated iteration over a collection by placing responsibility for iteration in a method on the collection. The method takes a Command object that is applied to the elements of the collection.

Programming languages already implement Enumeration Method as their loop construct. (e.g. *.forEach()*)



Collection
Type
enumerationMethod(Command)
Command
Type
executeOn(Type)
Loop administration is handled in the implementation of the enumerationMethod
Loop body is now provided as the implementation of the executeOn method

3.3.3 Vorteile

- Client is not responsible for loop housekeeping details
- Synchronization can be provided at the level of the whole traversal rather than for each element access
- Kann auch die Vorteile des Command-Patterns haben

3.3.4 Nachteile

- Functional approach, more complex syntax needed
- Often considered too abstract for programmers
- Hebelt Command Objekt aus

3.4 Batch Method

3.4.1 Problem

- Collection and client (iterator user) are not on the same machine
- Operation invocations are no longer trivial

Intent:

- How can a collection be iterated over multiple tiers without spending far more time in communication than in computation?

3.4.2 Solution

Group multiple collection accesses together to reduce the cost of multiple individual accesses in a distributed environment.

- Define a data structure which groups interface calls on client side
- Provide an interface on servant to access groups of elements at once

3.4.3 Eigene Notizen

Beispiel: StringBuilder. Die toString()-Methode wäre dann die Batch-Method

3.5 States

- Objects for States → Resultiert in vielen Klassen und Strukturen
- Methods for States → Propagiert eine einzelne Klasse mit vielen Methoden
- Collection for States → Ermöglicht mehrere State Machines mit der selben Logik zu managen und splittet Logik und Transaction Management in 2 Klassen

3.6 Objects for State

3.6.1 Problem

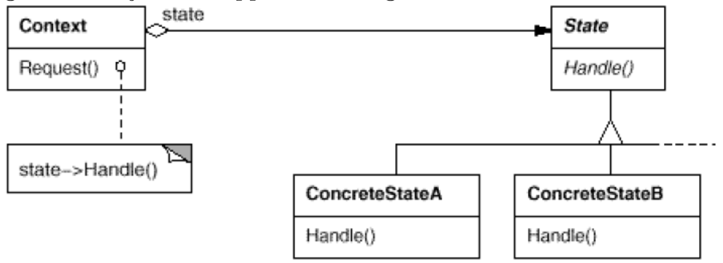
- Object's behaviour depends on its state, and it must change its behaviour at run-time
- Operations have large, multipart conditional statements (Flags) that depend on the state

Intent:

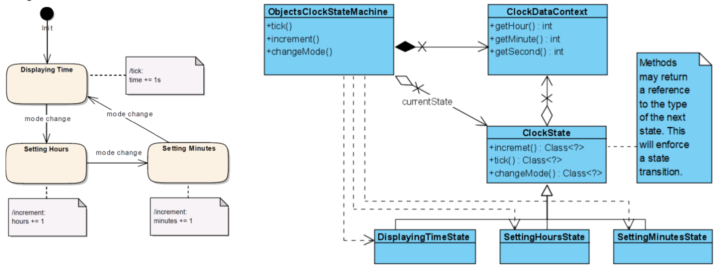
- How can an object act according to its state without multipart conditional statements?

3.6.2 Solution

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.



Dynamic:



3.6.3 Kritik

- Complex, but does not adequately cover that complexity
- Overkill in many cases
- Name suggests problem domain rather than the solution

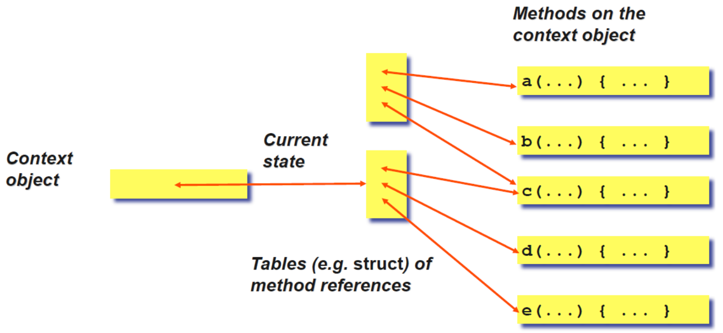
3.6.4 Eigene Notizen

- Objects for State heisst, dass pro State ein Objekt erstellt wird
- Ist im GoF als State-Pattern beschieben

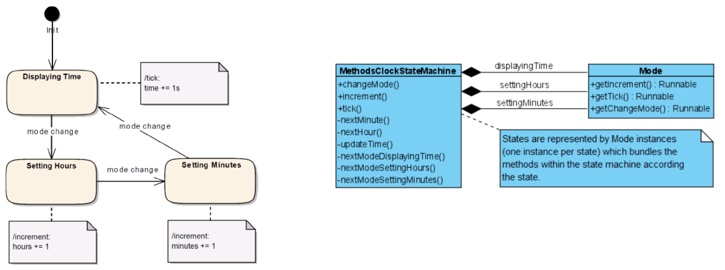
3.7 Methods for State

3.7.1 Solution

- Each state represents a table or record of method references
- The methods reference lie on the State Machine (context) object



Dynamics:



3.7.2 Vorteile

- Allows classes to express different behaviours in ordinary methods themself
- Behaviour coupled to the state machine, not scattered accross small classes
- Each distinct behaviour is assigned its own method
- No object context needs to be passed around, methods can already access the internal state of the State Machine

3.7.3 Nachteile

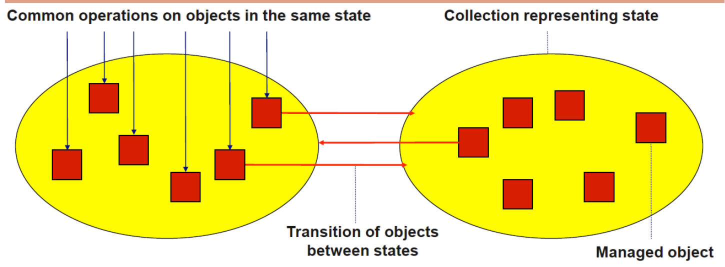
- Requires an additional two levels of indirection to resolve a method call
- The state machine may end up being far longer than was intended or is manageable

3.8 Collection for State

3.8.1 Beschreibung

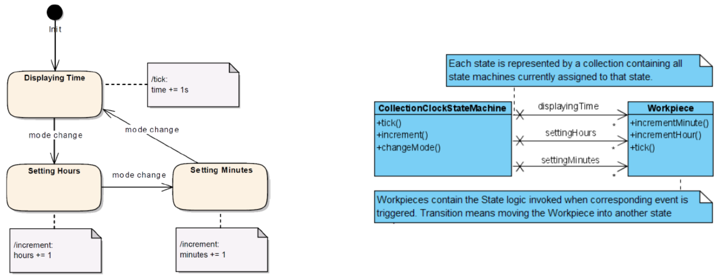
Jeder Zustand wird von einer Collection repräsentiert, die alle State Machines enthaltet.

3.8.2 Solution



Common operations on objects in the same state
Collection representing state
Transition of objects between states
Managed object

Dynamics:



Init
Displaying Time
/tick: time += 1s
mode change
Setting Hours
mode change
Setting Minutes
/increment: hours += 1
/increment: minutes += 1
Each state is represented by a collection containing all state machines currently assigned to that state.
CollectionClockStateMachine
+tick()
+increment()
+changeMode()
displayingTime
settingHours
settingMinutes
Workpiece
+incrementMinute()
+incrementHour()
+tick()
Workpieces contain the State logic invoked when corresponding event is triggered. Transition means moving the Workpiece into another state

3.8.3 Vorteile

- No need to create a class per state
- Optimized for multiple objects (state machines) in a state
- Object's collection implicitly determines its state → Zustand muss nicht intern repräsentiert werden
- Can be combined with other state machine (Objects/Methods)

3.8.4 Nachteile

- Can lead to a more complex state manager

3.9 Implementation of State Patterns

Objects:

- Results in a lot of classes and structures
- At least one class per state plus state machine

Methods:

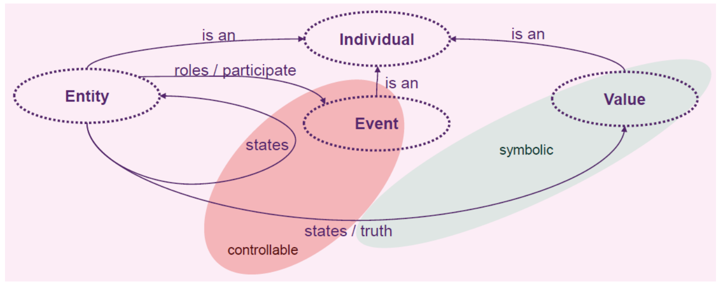
- Propagates a single class with a lot of methods

Collection:

- Allows to manage multiple state machines with the same logic
- Splits logic and transaction management into two classes

4 System Analysis

4.1 Individuals



Events:

- Individual Happening, taking place at some particular point in time
- *E.g. User-Action*

Entities:

- Individual that persists over time and can change its properties and states.
- May initiate events
- May cause spontaneous changes to their own states
- May be passive
- *E.g. Person*

Values:

- Intangible (nicht greifbar) individual that exists outside time and space
- Not subject to change
- *E.g. Körpergrösse*

5 Software Design

5.1 Categories of Objects

Entity

- Express system information
- Typically persistent in nature
- Identity is important to distinguish entity objects
- *E.g. User Klasse*

Service

- Represent system activities
- Distinguished by their behaviour rather than their state
- *E.g. User Service*

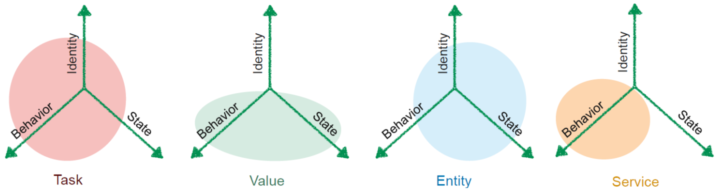
Values

- Interpreted content is the dominant characteristic
- Transient and do not have significant enduring indentity
- *E.g. Feld Alter, welches als Integer abgebildet wird*

Task

- Represent system activities
- Have an element of identity and state
- *E.g. Threads, Command Objects*

6 Object Aspects

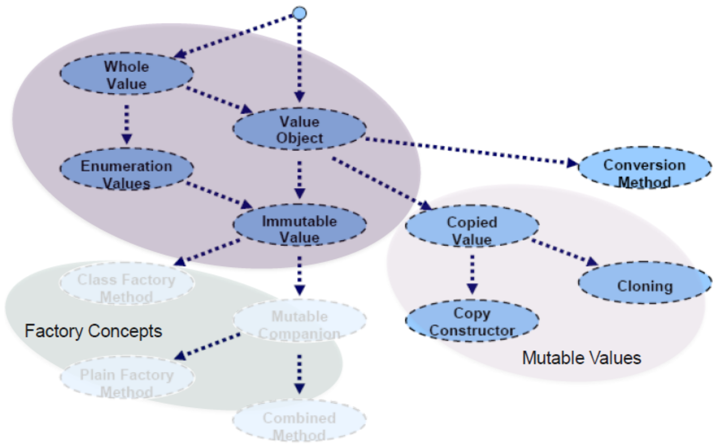


- Identity: Identität bedeutsam, oder vergänglich?
- State: Objekt stateful, oder stateless?
- Behavior: Hat das Objekt ein signifikantes Verhalten unabhängig von seinem State?

7 Value Objects

- Usually do not appear in UML class diagrams (except attribute type)
- Model fine-grained information
- Contain repetitive common code
  - U sed to add meaning to primitive value types
- Ensure type safety
- *E.g. IBAN Type Object (10 digits, checksum)*

8 Value patterns



8.1 Whole Value

8.1.1 Problem

- Plain integers and floating-point numbers are not very useful as domain values
- Errors in dimension and intent communication should be addressed at complile time
- How can you represent primitive quantities from your proble domain without loss of meaning?

8.1.2 Solution

Express the type of the quantity as a Value Class.

- Recovers the loss of meaning and checking by providing a **dimension** and **range**
- Wraps simple types or attribute stes
- Disallows inheritance to avoid slicing
- *E.g. Year, Month, Day Classes for Dates*

```
public final class Date {
    public Date(Year year, Month month, Day day) { ... }
}
```

8.2 Value Object/Value Class

8.2.1 Problem

- Comparison, indexing and ordering should not rely on objects identity but its content
- How do you define a class to represent values in your system?

8.2.2 Solution

Override methods in Object whose action should be related to content and not identity and implement serializable.

- Override Object's methods who define equality
- Java
  - b oolean equals(Object other)
  - i nt hashCode()
  - i mplement Serializable/toString() if appropriate
- TypeScript
  - e quals(other: Object): boolean
  - i mplement toString() if appropriate
- Overriding toString() can help using your Value Object

8.3 Conversion Method

8.3.1 Problem

- Values are strongely informational objects without a role for separate identity
- Often Value Objects are somehow related but cannot be used directly without conversion
- How could you use different, related Value Objects together without depending on underlying primitive type?

8.3.2 Solution

Provide type conversion methods responsible for converting Value Objects into related formats.

- Provide a constructor which converts between types
  String(char[ ] value)
- Or create a conversion instance method that converts to other type
  Date.toOtherType()
- Or create a **Class Factory Method** with conversion characteristics
  Date.from(Instant i) → static

8.4 Immutable Value

8.4.1 Problem

- A value exists outside time and space and is not subject to change
- Avoid side effect problems when sharing Value Objects
- Sharing values across Threads requires thread safety
- Values are often threaded as key for associative tables
- How can you share Value Objects and guarantee no side effect problems?

8.4.2 Solution

Set the internal state of the Value Class object at construction and allow no modifications.

- Declare all fields private final
- Mark class as final
- No Syncrhonization needed

8.5 Enumeration Values

8.5.1 Problem

- A fixed range of values should be typed *e.g. months*
- Using just int constants doesn't help
- Whole Value is only half the solution; range should be constant
- How can you represent a fixed set of constant values and preserve type safety?

8.5.2 Solution

Treat each constant as Whole Value instance declaring it public.

- Implement a Whole Value and declare the Enumeration Values as *public readonly* fields
- Prevent inadvertently changing the constants
- Pattern is built in (enum)

8.6 Copied Value and Cloning

8.6.1 Problem

- Values should be modifiable without changing the origins internal state
- How can you pass a modifiable Value Object into and out of methods without permitting callers or called methods to affect the original object?

8.6.2 Solution

Implement Cloneable interface to be used whenever a value object needs to be returned or passed as a parameter.

- Clone every Value Object leaked across boundaries (parameters / return values)
- May result in immense object creation overhead (cloning is expensive)
- Imitates *call-by-value* and *return-by-value*

8.7 Copy Constructor

8.7.1 Problem

- Within Value Objects we often know exactly what to copy
- How can objects be copied without the need of implementing a clone method?

8.7.2 Solution

Provide a way to construct an object based on another instance with exactly the same type.

- Declare the class *final* and derive from Object only
- Create a copy constructor, which consumes an instance with same type

```
public final class Date {
    public Date(Date other) {
        // ...
        this.year = new Year(other.year);
    }
}
```

8.8 Class Factory Method/Simple Factory

8.8.1 Problem

- Construction of Value Objects may be expensive
- Different construction logic is required which may result in huge amount of constructors
- How can you simplify and potentially optimize construction of Value Objects in expressions without introducing new intrusive expressions?

8.8.2 Solution

Provide static methods to be used instead of ordinary constructors. The methods return either newly created Value Objects or cached Objects.

- Declare one or more creation method on the class
- Define constructors *private*, they are invoked by Class Factory Method
- The static methods could also contain caching mechanisms

```
public final class Year {
    public static Year of(int value) {
        return new Year(value);
    }
    private Year(int value) { this.value = value; }
}
```

8.9 Mutable Companion

8.9.1 Problem

- We need to calculate with alues *e.g. 15 working days after exam*
- How can you simplify complex construction of an immutable value?

8.9.2 Solution

Implement a companion class that supports modifier methods and acts as a factory for immutable Value Objects.

- Factory Object for immutable values
- Neither a subtype nor a supertype of the Immutable Value
- In combination with companion, the factory is called Plain Factory Method

```
public final class YearCompanion {
    private int value;
    public YearCompanion(Year toModify) { this.value =
        toModify.getValue(); }
    /* modifying methods */
    public Year asValue() { /* Factory Method */
        return Year.of(value);
    }
}
```

8.10 Relative Values

8.10.1 Problem

- Value Objects are compared by their state, not their identity
- Relative comparison between Value Objects for appropriate values should be provided
- How can a Value Obejct be compared against others in a typed way?

8.10.2 Solution

Implement the responsibility for comparison between Value Objects by deriving from *Comparable* interface.

- Override-Overload Method Pair: Comparable? and equals(? other) should be overridden as pair
- TypeT Specific Overload: Implement ComparableT .compareTo(T other) and equals(T other)
- Bridge Method: Provide a Method for *equals(Object other)* and forward it to *equals(T other)*

8.11 Discussion

Difference between Whole Value and Immutable Value

- Whole: Value with a unit should be encapsulated in a class
- Immutable: Value within an object must be immutable

When would you prefer Class Factory Method over a conversion constructor?

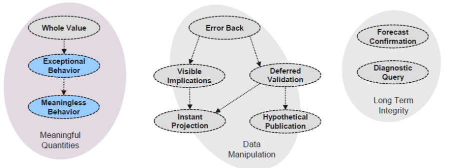
- Factory: A foreign value type should be converted into the current value format
- Contructor: A more generic type should be converted into the current type

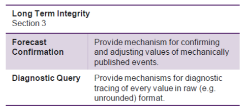
What are the most important liabilities of the Mutable Value concept?

- Concept of Cloning/Copied Value may be missed by the programmer which results in to hard to find errors

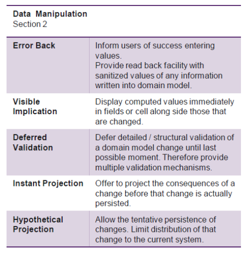
9 Checks

- Separate good input from bad
- Information integrity checks
- Applied without complicating the program





Long Term Integrity Section 3	
Forecast Confirmation	Provide mechanism for confirming and adjusting values of mechanically published events.
Diagnostic Query	Provide mechanisms for diagnostic tracing of every value in raw (e.g. unrounded) format.



Data Manipulation Section 2	
Error Back	Inform users of success entering values. Provide read back facility with sanitized values of any information written into domain model.
Visible Implication	Display computed values immediately in fields or cell along side those that are changed.
Deferred Validation	Defer detailed / structural validation of a domain model change until last possible moment. Therefore provide multiple validation mechanisms.
Instant Projection	Offer to project the consequences of a change before that change is actually persisted.
Hypothetical Projection	Allow the tentative persistence of changes. Limit distribution of that change to the current system.

9.1 Exceptional Behaviour

9.1.1 Problem

- Missing or incorrect values in a domain model are impossible to avoid
- The domain logic should be able to handle this sort of missing data
- How can exceptional behaviour caused by invalid input be handled without throwing errors?

9.1.2 Solution

Use one or more distinguished values to represent exceptional circumstances.

- Invalid parametrized domain calls may produce Exceptional Values
- Domain logic may accept Exceptional Value as legal input

```
// TypeScript
export class Calculator {
  public static divide(numerator: number, divisor: number):
      number | CalculationError {
    if (divisor === 0) { return CalculationError.DivByZero; }
    if (isNaN(numerator)) { return
        CalculationError.NumeratorIsNaN; }
    if (isNaN(divisor)) { return
        CalculationError.DivisorIsNaN; }
    return numerator / divisor;
  }
}
```

9.2 Meaningless Behaviour

9.2.1 Problem

- Due to error handling, domain logic may be expressed with more complexity than originally conceived
- How can exceptional behaviour due to invalid input be handled without throwing errors?

9.2.2 Solution

Write methods with minimalistic concern for possible failure.

- Initiate computation
- If it fails:
  - r ecover from failure and continue processing
  - e nsure the error is logged/visualized on surface
- Choose meaningless behaviour unless a condition has domain meaning
- Represents an alternative implementation of Exceptional Value

```
// TypeScript
export class Calculator {
  public static divide(numerator: number, divisor: number):
       number {
  return numerator / divisor; // may result in Infinity (
       Java: NaN) if divisor is 0.0
  }
}
```

10 Framework Introduction

Why Frameworks

- Avoid re-inventing the wheel
- It is easy but inefficient to program the same thing again and again

What is a Framework

- Object-Oriented classes that work together → Many design patterns are microframeworks
- Framework provides *hooks* for extension
- In contrast to a library, a framework keeps the control flow, not your extension → Inversion of control via callbacks

Framework Callbacks

- Hollywood Principle: Don't call us, we call you! → Control flow is from framework to application components
- Extendability and configurability

10.1 Application Framework

- Object-oriented class library
- *Main()* program lives in the Application Framework
- Provides *hooks* and *callbacks*
- Provides ready-made classes for use
- Creates product families
- Reuse of application architecture and infrastructure

10.2 Examples

Frameworks

- .NET Core
- Entity Framework
- React (lib)
- Vue

Application Framework

- Spring
- ASP.NET
- Angular

10.3 Difference: Library / Framework / App Framework

Library

- Contain 3rd party Features which do not control the application flow (e.g. Math Library)

Framework

- Provide Hooks / Extension points
- Strogly rely on Inversion of Control (IoC) principle →
- The framework defines when hooks (activities) are called thus controls part of the application flow
- E.g. JRE, .NET Fx, Vue, React

App Framework

- Contains the *main()* procedure
- Clients implement plugins or components defined as extension points by the framework
- Completely controlls the app flow
- E.g. Spring Boot, ASP.NET, Angular

10.4 Summary

Benefits

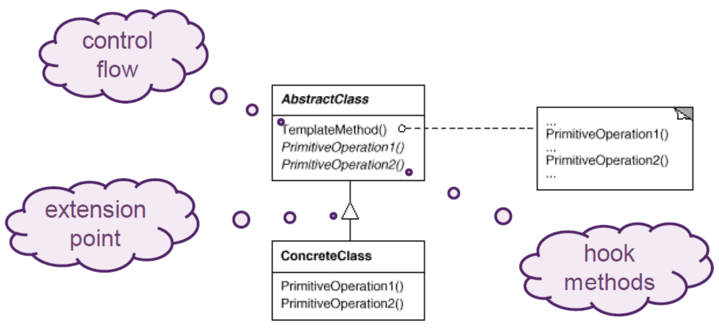
- Less code to write
- Reliable and robust code
- Consistent and modular code
- Reusability
- Maintenance

Liabilities

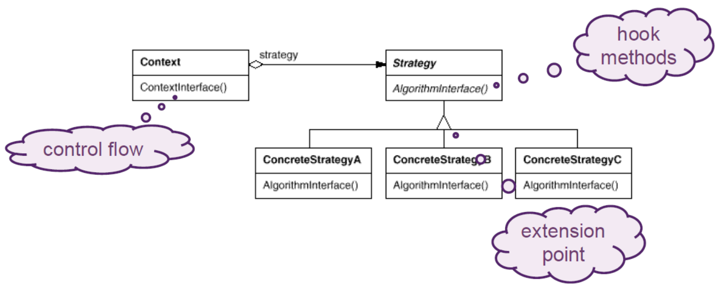
- Portability: Code is strongly coupled to the overlying Framework
- Testing: Close coupling between framework parts
- Evolution: User's implemetation may break due next Framework version

10.5 Hook / Extension Point / Control Flow

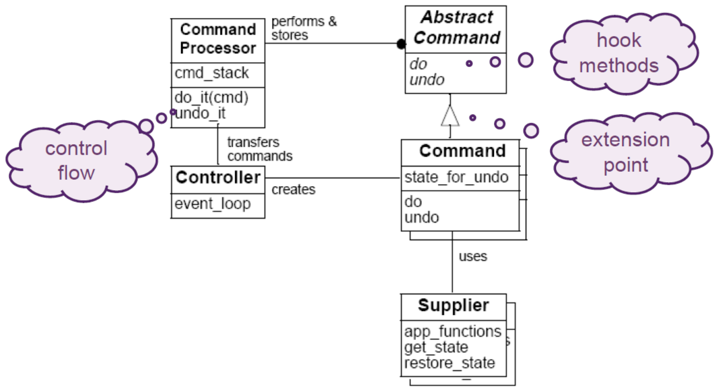
10.5.1 Template Method



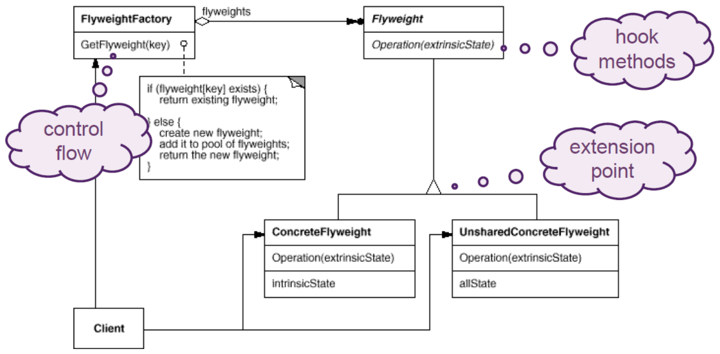
10.5.2 Strategy



10.5.3 Command Processor



10.5.4 Flyweight

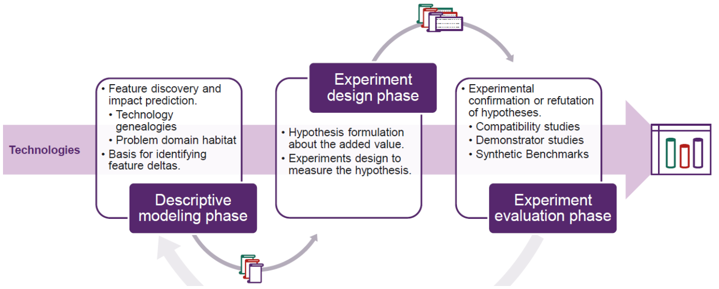

FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
hook methods
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
control flow
extension point
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

11 Meta Frameworks

A Framework for Evaluating Software Technology

- Initial acquisition cost
- Long-term effect on quality, time to market, and cost of the organization's products and services
- Training and support
- Relationship to future technoloy plans
- Response of direct competitor organizations

11.1 Framework Evaluation Phases


Technologies
Feature discovery and impact prediction.
Technology genealogies
Problem domain habitat
Basis for identifying feature deltas.
Descriptive modeling phase
Experiment design phase
Hypothesis formulation about the added value.
Experiments design to measure the hypothesis.
Experimental confirmation or refutation of hypotheses.
Compatibility studies
Demonstrator studies
Synthetic Benchmarks
Experiment evaluation phase

12 Developing Frameworks

- Frameworks need evolutionary improvements

12.0.1 Frameworkers Dilemma

Potential ways out of the dilemma

1. Think very hard up-front
2. Don't care too much about framework users

3. Let framework users participate
4. Use helping technology

13 Meta Pattern

- Reflection is often used as technology for Meta Programming
- Provide Flexibility, Adaptability & Generality

Recurring Problems

- Reflection solves common frameworkers problems
- Exchanging parts of a software system is hard
- Not yet unknown sofware components should be integrated

13.1 Reflection

Usage (Java/C#)

- Load of JAR/DLL files at runtime
- Invoke Methods
- Read out properties/fields
- Create object instances
- Search for annotations on Classes / Methods / Fields / ...

Provides Facility to implement (Usage):

- DI
- Convention over Configuration
- Object-Relation Mapper
- Serialization/Deserialization
- Plugin Architectures

Consists of two aspects:

Introspection

- The ability for a program to observe and therefore reason about its own state
- e.g. Query object properties, get list of methods

Intercession

- The ability of a program to modify its own execution state or alter its own interpretation of meaning
- e.g. Modify object properties, add another attribute or exchange code

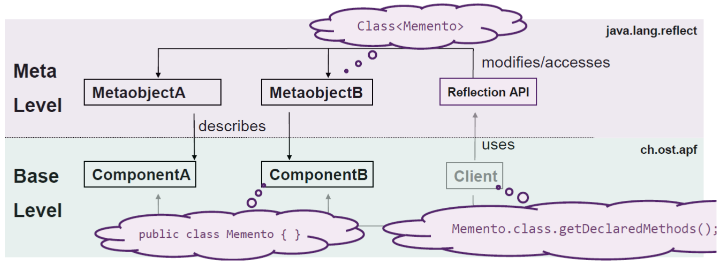
Defines meta level and base level:

Meta level

- Provides self-representation
- Gives the SW a knowledge of its own structure
- Consists of meta objects

Base level

- Defines the app logic
- Implementation may use the meta objects



13.1.1 Summary

Benefits

- Adapting a software system is easy
- Support for many kinds of changes

Liabilities

- Produces non-transparent APIs
- Binding at runtime (limited Type safety)

Dangers

- Different mechanisms for similar semantics confuse
- More indirection costs efficiency - sometimes too much
- Too much work to configure and instantiate systems
- Obscure APIs
- Overengineered solutions
- Security mechanisms get in the way or are undermined

13.2 Meta Objects

Languages using reflections use meta objects for:

- Classes
- Object Attributes
- Methods
- Class Relationships

13.3 Type Object

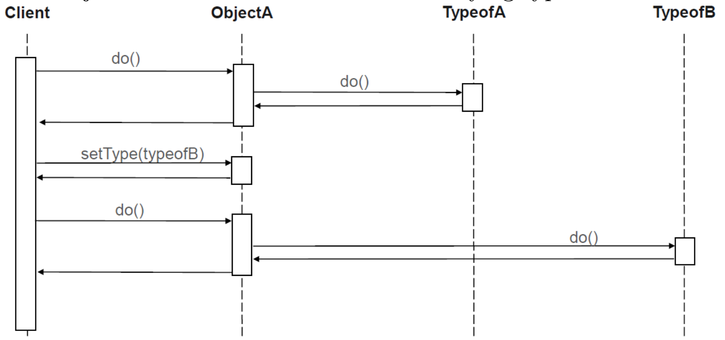
13.3.1 Problem

- We want to keep common behaviour and data in only one place
- DRY implementation of domain
- How can you categorize objects, eventually dynamically?

13.3.2 Solution

Categorize objects by another object instead of a class. Thus an object can change it's 'class' at runtime

- Create a category (type) object which describes multiple objects
- Objects forward the calls to the underlying type



13.3.3 Summary

Benefits

- Categories can be added easily, event at runtime
- Avoids explosion of (trivial) subclasses
- Allows multiple meta-levels

Liabilities

- Confusing mess of 'classes' because of separation
- Lower efficiency because of indirection
- Changing databse schemas can be tricky

13.3.4 Discussion

Based on which GoF Pattern

- Strategy

Similar intent in a GoF Pattern

- State, also changes at runtime

13.4 Property List

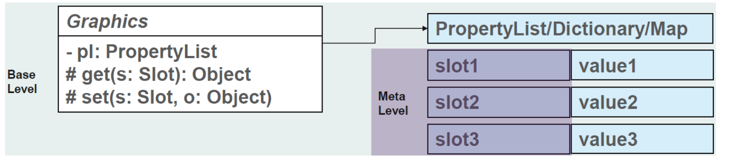
13.4.1 Problem

- Attributes should be attachable/detachable after compilation
- Objects share attributes/parameters across the class hierarchy
- How do you define properties in a flexible way so they can be attached and detached at runtime?

13.4.2 Solution

Provide objects with a property list. That list allows to associate names with other values or objects.

- Property list maps attribute names to values
- each name defines a slot
- Same slot can be used for attributes with identical semantics
- Objects can be triggred to list all slot names and values



13.4.3 Summary

Benefits

- Attributes can be added dynamically
- Object extension while keeping object identity
- Easy attribute iteration

Liabilities

- Different ways to access regular/dynamic attributes
- Type safety left to the programmer
- Naming not checked by a compiler
- Run-time overhead
- Memory Management

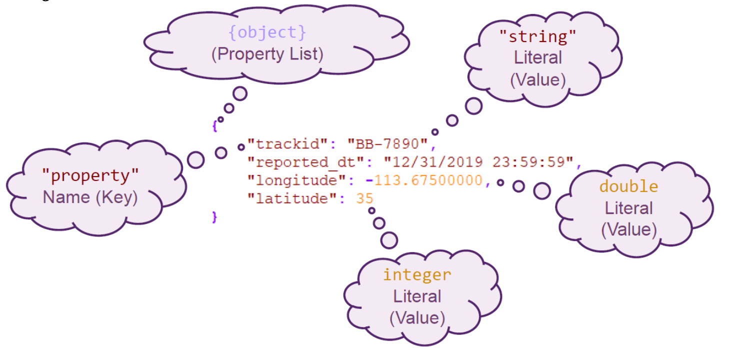
Mitigate Liabilities: Bridge Methods

13.5 Anything

What is Anithing?

- Arbitrary data structure
- Recursively structured Property List
- Internal representation of today's JSON

Object Definition:



```
{
 "trackid": "BB-7890",
 "reported_dt": "12/31/2019 23:59:59",
 "longitude": -113.67500000,
 "latitude": 35
}
```

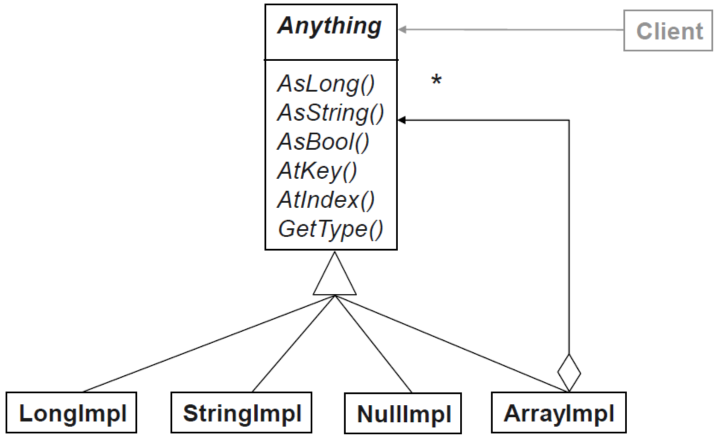
13.5.1 Problem

- We need to keep a map of data similar to the Property List
- Structured data also includes sequences of data
- Data should be structured recursively
- How do you provide a generic configuration or communication data structure that is easily extensible?

13.5.2 Solution

Create an abstraction for structured values that is self describing.

- Implement a representation of simple values
- Add an implementation for a sequence of values (& key value access)
- Provide a default value if requested value cannot be converted



13.5.3 Summary

Benefits

- Readable streaming format
- Appropriate for configuration data
- Universally applicable
- Flexible interchange across class/object boundaries

Liabilities

- Less type safety
- Intent of parameter elements not always obvious
- Overhead for value lookup and member access
- No real object, just data

13.5.4 Discussion

Which GoF Pattern does Anything implement?

- Transparent Composite Pattern
- Null Object

14 Boxing / Killing

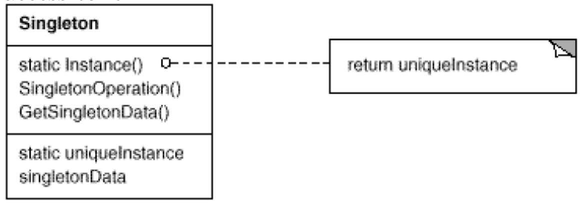
14.1 Singleton Boxing/Killing

14.1.1 Problem

- Some Classes should have only one instance
- The instance must be accessible from a well-known access point
- Subclassing from the Singleton should be possible
- Extending the Singleton class must not break existing code
- How can be guaranteed that only one object of a class is instantiated and can globally be accessed?

14.1.2 Solution

Ensure a class only has one instance and provide a global point of access to it



14.1.3 Solutions inside Singleton

- Singleton Pattern
- Class Factory Method
- Lazy Acquisition
- Eager Acquisition

14.1.4 Implementation

```
public class Singleton {
  private static class InstanceHolder {
  // Singleton will be instantiated as soon as the
       ClassLoader instantiates the overlying class.
  private static final Singleton INSTANCE = new Singleton();
  }
  public static Singleton getInstance() {
    return InstanceHolder.INSTANCE;
  }
  protected Singleton() { } // allow subclassing
}
```

14.1.5 Summary

Benefits

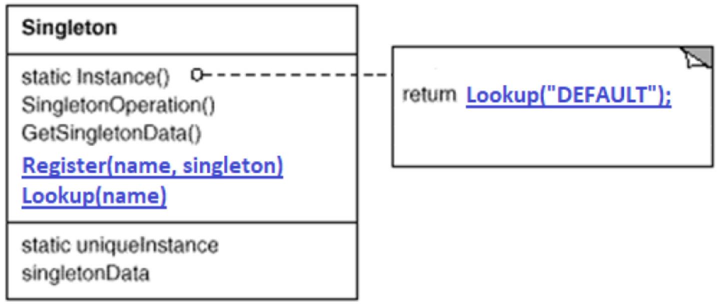
- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations and representation
- Permits variable number of instances
- More flexible than class operations

Liabilities

- Introduces a global variable/state
- Prevents polymorphism
- Carries state until app closes
- Restricts unit testing due limited interchangeability

14.2 Singleton Variation 'Registry'

- More flexible approach
- Uses a registry of singletons
- Classes registers their singleton in a well-known registry



14.3 Monostate (Borg) - Killing

14.3.1 Problem

- Multiple instances should have the same behaviour
- The instances should be simply different names for the same object
- Should have the behaviour of Singleton without imposing the constraint of a single instance
- How can two instances behave as though they were a single object?

14.3.2 Solution

Create a monostate object and implement all member variables as static members

```
// Example 1
public class Monostate {
  private static int x;
  private static int x;
  public int getX() { return x; }
  public int getY() { return y; }
}
// Example 2
public interface Monostate {
  int getX();
  int getY();
}
public class MonostateImpl implements Monostate {
  public int getX() {
    return Singleton.getInstance().getX();
  }
  public int getY() {
    return Singleton.getInstance().getY();
  }
}
```

14.3.3 Summary

Benefits

- Transparency, no need to know about Monostate
- Derivability, subclasses allowed
- Polymorphism, different derivates can offer different behavior
- Well-defined creation and destruction

Liabilities

- Breaks inheritance hierarchy
- Memory usage, because statics are always allocated
- Unable to share Monostate across several tiers

14.4 Service Locator

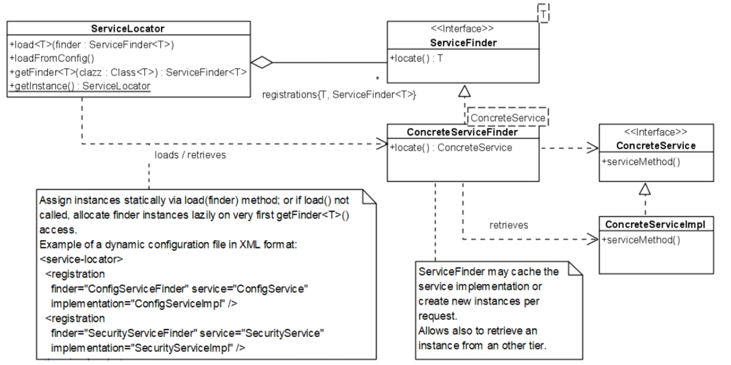
14.4.1 Problem

- Implementation of a global service instance should be exchangeable
- It should be possible to execute the service methods on another tier transparently
- How could we register and locate global services when one is needed?

14.4.2 Solution

Implement a service locator that knows how to hold all of the services that an application might need.

- Implement the ServiceLocator as a Singleton 'Registry'
- ServiceLocator returns *finder* instances, which are used to locate the underlying services
- Also known as dynamic ServiceLocator or Provider

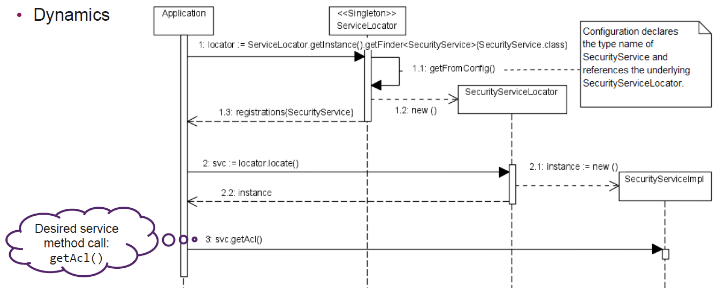


Assign instances statically via load(finder) method; or if load() not called, allocate finder instances lazily on very first getFinder<T>() access.
Example of a dynamic configuration file in XML format:

```
<service-locator>
  <registration
   finder="ConfigServiceFinder" service="ConfigService"
   implementation="ConfigServiceImpl" />
  <registration
   finder="SecurityServiceFinder" service="SecurityService"
   implementation="SecurityServiceImpl" />
```

ServiceFinder may cache the service implementation or create new instances per request.
Allows also to retrieve an instance from an other tier.

• Dynamics


Application
<<Singleton>> ServiceLocator
1: locator := ServiceLocator.getInstance().getFinder<SecurityService>(SecurityService.class)
1.1: getFromConfig()
Configuration declares the type name of SecurityService and references the underlying SecurityServiceLocator.
SecurityServiceLocator
1.2: new ()
1.3: registrations{SecurityService}
2: svc := locator.locate()
2.1: instance := new ()
SecurityServiceImpl
2.2: instance
Desired service method call: getAcl()
3: svc.getAcl()

14.4.3 Summary

Benefits

- There is exactly ONE Singleton in the application
- ServiceLocator interface strogly rely on abstractness

Liabilities

- Clients stll rely on a static reference to ServiceLocator class (tight coupling)
- No possibility to replace the ServiceLocator

14.5 Parameterize from Above

- Singleton pattern doesn't provide durability and testability requirements
- The application has been layered (horizontally) in a logical manner
- How can we povide the required application-wide data to the lower layers without making the data globally accessible?

14.5.1 Solution

Parameterize each layer from above. Data that affect the behaviour of lower layers should be passed in from the top of the stack.

- Pass in configuration parameters and 'known' objects rather than having them global

14.5.2 Summary

Benefits

- No global variables
- Implementations of parameterized functionalities are exchangeable
- Enforces separation-of-concenrns at architecture level
- Reduces coupling between layers

Liabilities

- Adds more complexity to the system
- Contexts must be passed through the whole app stack
- Fragile Bootstrapper: app must be wired completely at startup

14.6 Dependency Injection

14.6.1 Problem

- User may override implementations of existing app components (e.g. test doubles)
- Any Componet within the system can demand an object of a specified interface
- The Components should not know anything about the wiring mechanism
- How can framework users provide their own implementation/specialization of predefined framework components and even extend the system by own interface definitions and implementations?

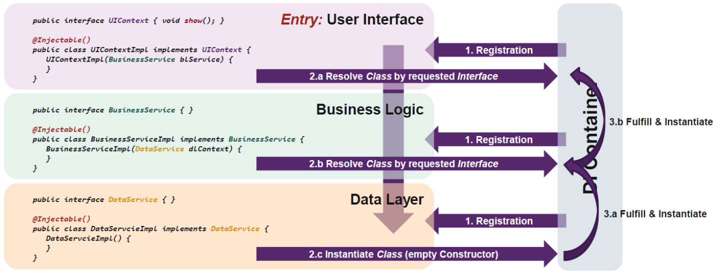
14.6.2 Solution

Introduce a DI Container which loads the interfaces and implementation calsses at startup and dynamically instantiates and wires the objects according to the dependency tree.

- Central container class (acts as Registry)
- Users reference dependencies by the required interfaces (resolved by container)
- Users apply code annotations
- Clients should not address the container directly

14.6.3 Implementation

- Combine *Service Locator* and *PfA*
- Container class must not be statically referenced by clients



```
public interface UIContext { void show(); }

@Injectable()
public class UIContextImpl implements UIContext {
   UIContextImpl(BusinessService blService) {
   }
}
```

```
public interface BusinessService { }

@Injectable()
public class BusinessServiceImpl implements BusinessService {
   BusinessServiceImpl(DataService dlContext) {
   }
}
```

```
public interface DataService { }

@Injectable()
public class DataServcieImpl implements DataService {
   DataServcieImpl() {
   }
}
```

Entry: User Interface
Business Logic
Data Layer
1. Registration
2.a Resolve Class by requested Interface
2.b Resolve Class by requested Interface
2.c Instantiate Class (empty Constructor)
3.b Fulfill & Instantiate
3.a Fulfill & Instantiate
DI Container

14.6.4 Summary

Benefits

- Reduces coupling between consumer and implementation
- Contracts between classes are based on interfaces
- Suports open/closed principle
- Allows flexible replacement of an implementation

Liabilities

- Adds black magic to the system
- Debugging the object dependency tree may become hard
- Recursive dependencies are hard to find and may prevent the sytem from startup
- Relies on reflection and can result in a performance hit

14.6.5 Discussion

Relation Singleton - DI

- Some injected Dependencies may be singletons
- DI Container implementation may be based on 'registry' singleton

Improvement of DI over Service Locator

- Client classes dont depend directly on the DI Container
- Less coupling between DI Container and Component

14.7 Flyweight

A single pattern for both **sharing** and **creation**

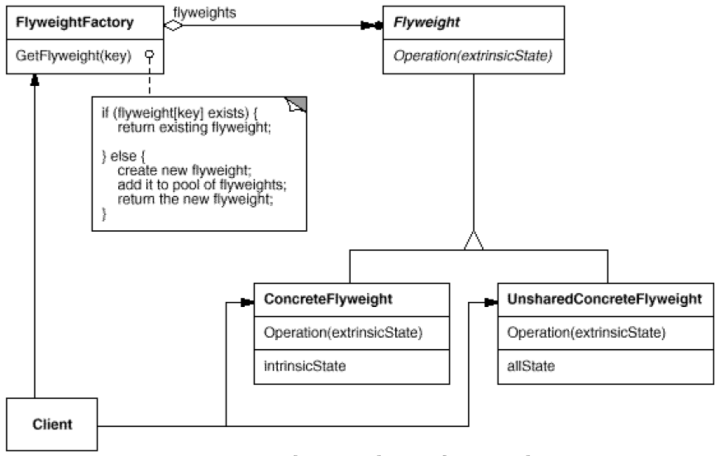
14.7.1 Problem

- Storage costs are high because of the sheer quantity of objects
- Many objects may be replaced by relatively few shared objects
- The objects do not depend on object identity
- How can multiple copies of a identical constant object (referenced objects) be avoided?

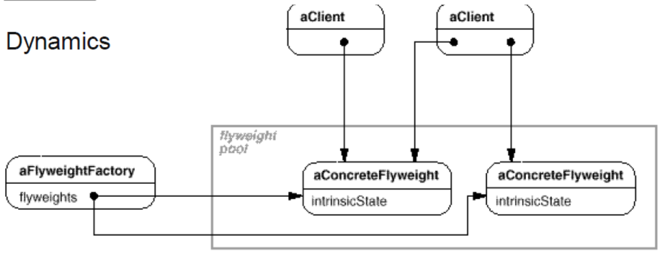
14.7.2 Solution

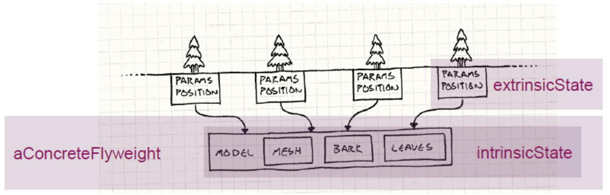
Use sharing to support large numbers of fine-grained objects efficiently

- Flyweight manager maintains instantiated flyweights
- Flyweights must be immutable (readonly)
- Context information is often maintained by parent object


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Dynamics





14.7.3 Solutions inside Flyweight

- Composite
- Immutable Value
- Pooling
- Class Factory Method
- Lazy Acquisition
- Eager Acquisition

14.7.4 Summary

Benefits

- Reduction of the total number of instances (space savings)

Liabilities

- Can't rely on object identity
- May introduce run-time costs

14.8 Pooling (Boxing)

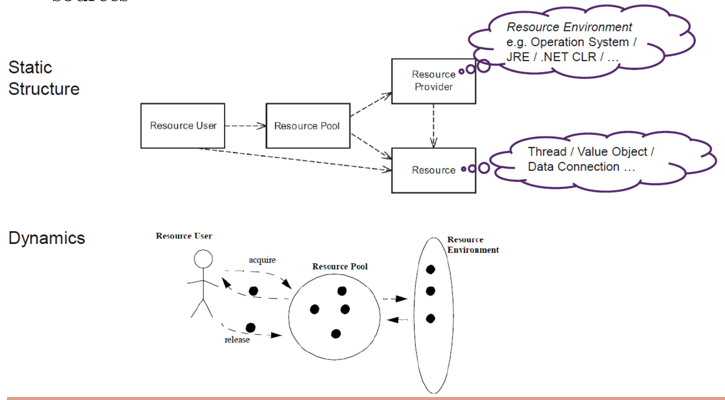
14.8.1 Problem

- A fast and predictable access to resources should be provided
- Wastage of CPU cycles in repetitious acquisition/release should be avoided
- Acquisition/release complexity should be minimized
- How can expensive acquisition and release of resources be avoided by recycling resources that are no longer needed?

14.8.2 Solution

Manage multiple instances of one type of resource in a pool. This pool of resources allows for reuse of released resources.

- A resource pool manages resources and gives them to the users
- Resource providers, such as OS, owns and manages the resources



14.8.3 Implementation

- Define the maximum number of resources that are maintained by the pool
- Decide between *eager* and *lazy* Acquisition
- Determine resources recycling/eviction semantics
  - E .g. clean up stack after thread execution has computed
  - U se appropriate allocation and destruction patterns

14.8.4 Summary

Benefits

- Performance of app
- Simplified release and acquisition of resources
- New resources can be created dynamically

Liabilities

- Certain overhead due resource management
- Acquisition requests must be synchronized to avoid race conditions

14.8.5 Discussion

Which pattern can be combined with Pooling, if the same resources should be shared between users?

- Pool acts as mediator

Relation of Flyweight and Pooling

- Flyweight implements a pool with immutable resources statically

Is immutability of named resources key?

- No

Difference between pooling and caching

- Caching is about handling resources with identity, pooling does not
- All resources in a pool are equal
- Caching only manages object lifetime in cache, not of objects themselve