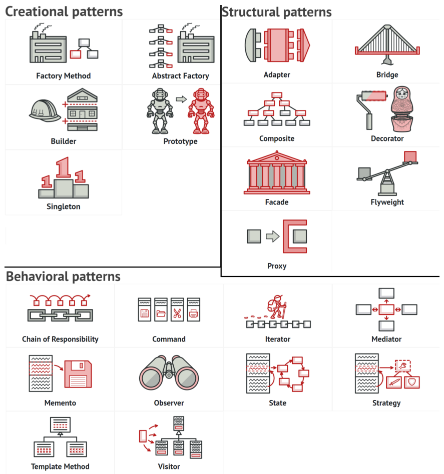
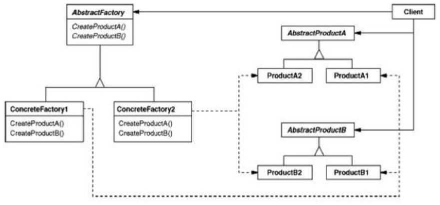
Pattern Overview



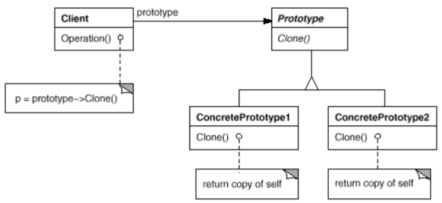
Creational

Abstract Factory



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

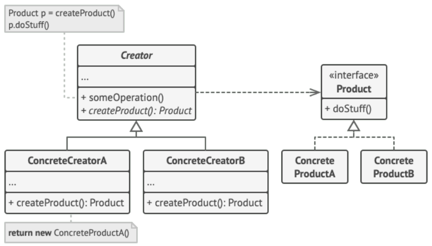
Prototype



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

- Does not describe how the manager is implemented

Factory Method



Intent: Abstract creation, Improve Testability
Problem: Creation interface on super, implementation in child
Solution: Factory method to defer insanitation to subclass (allows superclass to alter type of objects that will be created)
Implementation: Testing: override with Mock creation
Relations: Setting Context by (Template Method), Abstract Factory

Singleton

Intent: Guaranteeing that only one global object of class exists
Problem: Some classes should have only one instance
Solution: Private construction and static class factory method
Implementation: `static Instance()` can use Lazy initialization
Relations: Setting context by (Class Factory Method)

- Sole instance, variable number of instances, flexibility
- Global variable, tight coupling, prevents polymorphism

Registry (Singleton Mitigation)

Intent: Guaranteeing that only one global object of class exists
Problem: Some classes should have only one instance
Solution: Classes register their "Singleton" in a well-known **registry**
Implementation: Interface for contained "Singletons" for testability

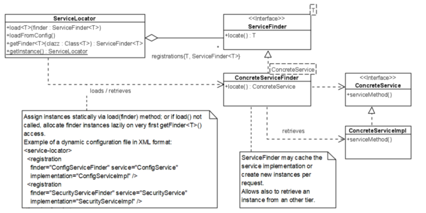
- More flexible than Singleton, Benefits of Singleton
- No object creation responsibility, Liabilities of Singleton

Monostate (Singleton Mitigation)

Intent: Multiple instances in same state and behavior
Problem: Multiple instances should have the same behavior
Solution: Monostate object: all member variables as static members
Implementation: Create monostate object and implement all member variables static
Relations: Specialization of (Singleton)

- Transparency, Drivability, Polymorphism, Testability
- Breaks inheritance hierarchy, no sharing,unexpected behavior

Service Locator (Singleton Mitigation)

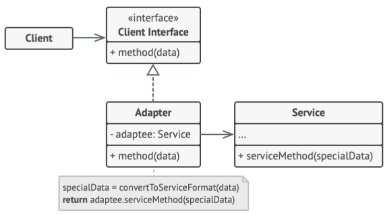


Intent: Manage dependencies and maintain loose coupling
Problem: Global service instance should be exchangeable
Solution: Locator locates services via Service Finders
Relations: Setting context by (Singleton)

- Only one Singleton, Abstract interface of ServiceLocator
- ServiceLocator still Singleton, cant replace ServiceLocator

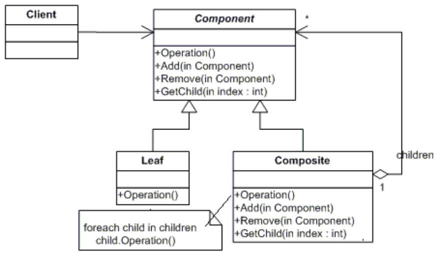
Structural

Adapter



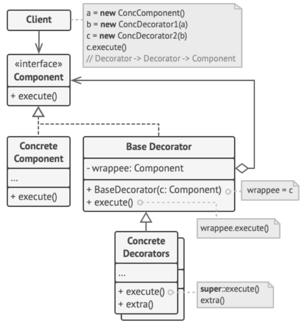
Convert the interface of a class into another interface expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Composite



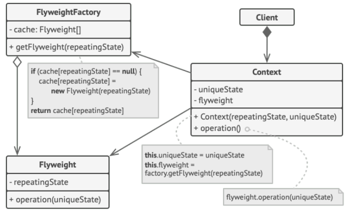
Composes objects into tree structures and let you work with these structures as if they were individual objects. Visitor

Decorator



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

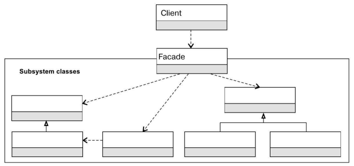
Flyweight



Intent: Avoid multiple copies of identical (constant/referenced) objects
Problem: Storage costs are high because the sheer quantity of objects
Solution: Use sharing to support large number of fine-grained objects
Implementation: Manager maintains Flyweight, Flyweights immutable
Relations: Setting Context by (Pooling), Composite

- Reduction of total Instances (Total number, state per object)
- Can't rely on object identity,Finding Flyweight maybe costly

Facade



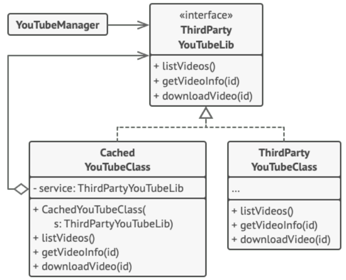
Provides a simplified interface to a library, a framework, or any other complex set of classes.

Pooling (Boxing)

Intent: Fast, predictable access to resource with minimal complexity
Problem: Slow and unpredictable access to resources
Solution: Manage multiple instances of one type in a pool, allows reuse
Implementation: Define max number of resources, Use lazy or eager
Relations: Setting Context for (Flyweight), acts as Mediator

- Performance,Lookup predictable,Simple,Dynamic allocation
- Management overhead, Synchronization to avoid races

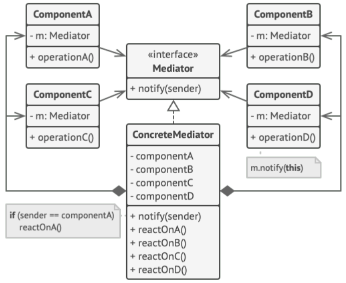
Proxy



Provide a surrogate/placeholder for another object to control access to it.

Behavioral

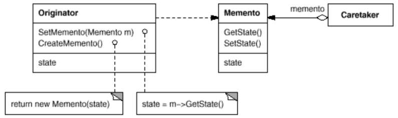
Mediator



Intent: Promotion of loose coupling, stops explicit referral
Problem: Strong coupling and complex communication
Solution: Introduce mediator to abstract interaction between objects
Implementation: Mediator as Observer, Colleages as subject
Relations: Refinement(Observer)

- Colleague classes may become more reusable, low coupling
- Centralizes control of communication between objects
- Encapsulates protocols
- Adds complexity Single point of failure

Memento (Vermittler)



Intent: It separates the serialization logic from the business logic.
Problem: Objects encapsulate their state, making it inaccessible, record internal state
Solution: Capture objects internal state in memento
Implementation: originator creates memento with internals

Participants

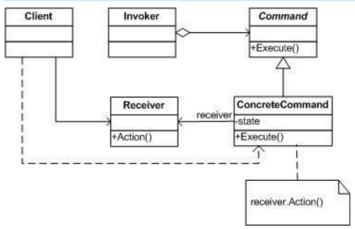
Memento
- Stores some or all internal state of the Originator
- Allows only the Originator to access its internal information

Originator
- Creates Memento objects to store its internal state at strategic points.
- Restores its own state to what the Memento object dictates

Caretaker (Filesystem, Database)
- Stores the Memento object of the Originator.
- Cannot explore the contents of or operate on the Memento object.

- Internal state of an object can be saved and restored at any time
- Encapsulation of attributes is not harmed
- State of objects can be saved later
- Creates a complete copy of the object every time, no diffs
- No direct access to saved state, it must be restored first

Command



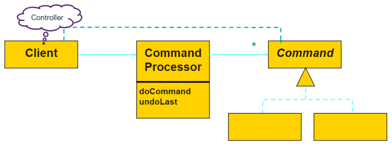
Encapsulates commands, so that they can be parameterized **scheduled**, **logged** and/or **undone**. => pattern does not describe history/undo management

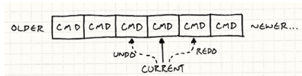
Intent: Encapsulation of commands so they can be scheduled/logged
Problem: Executed Methods not identifiable in most languages
Solution: Encapsulate request as object, used for parameterization
Relations: Setting Context for (Command Processor, Internal Iterator), Setting Context by (Strategy)

- 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
- Large designs with many commands can introduce many small command classes mauling the design

Command Processor

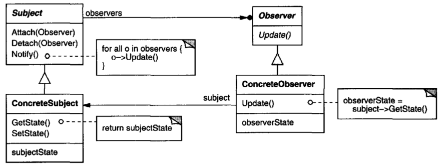
Intent: Manage command objects so execution can be undone
Problem: History and undoing of method execution impossible
Solution: Command Processor manages requests as objects
Implementation: Command Processor contains Command History
Relations: Setting Context by (Command, Memento)





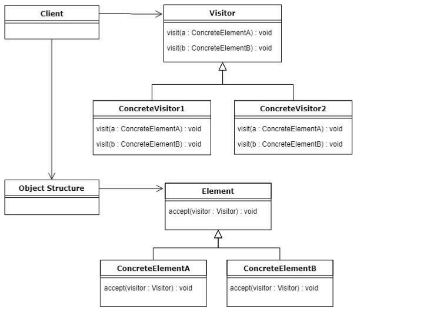
- Flexibility, Command Processor and Controller are implemented independently of Commands
- Central Command Processor allows addition of services related to Command execution
- Enhances testability, Command Processor can be used to execute regression tests
- Efficiency loss due additional indirection

Observer



Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Visitor



Intent: Separate algorithms from the objects on which they operate
Problem: Different algorithms needed to process composite tree
Solution: New behavior in separate class instead integrating in original
Implementation: Visitor interface with "visiting" methods e.g. accept()
Relations: Composite, (Iterator), Chain of Responsibility, Interpreter

Criticism

Visitor can be overused
- Put important stuff into node classes, see Interpreter

Visitor bad when visited class hierarchy changes
- Hard to change or adapt existing visitors

Keeping state during visitation
- State is shared within the visitor object
- But hard when different visitors need to collaborate

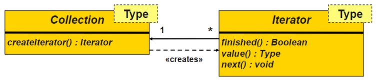
- Visitor makes adding new operations easy
- Separates related operations from unrelated ones
- Adding new node classes is hard
- Visiting sequence fix defined within nodes
- Visitor breaks logic apart

Iterator Patterns

Intent: Avoid strong coupling between collection and iteration
Problem: Iteration of collection depends on target implementation

External Iterator

Problem: Knowledge for iteration is separate object from target
Solution: Four operations: create, hasNext, (access, next)
Relations: Setting Context by (Plain Method Factory)

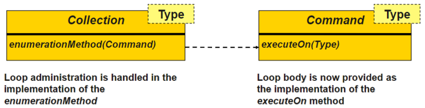


👍 Provides a single interface to loop through any kind of collection

👎 Multiple iterators may loop through a collection at the same time
👎 Life-cycle management of iterator objects
👎 Close coupling between Iterator and corresponding Collection class
👎 Indexing *might* be more intuitive for programmers

Internal Iterator

Solution: Responsibility for iteration on collection, uses Command
Implementation: Implemented by most Programming langs `.forEach)`
Relations: Setting Context by (Command, Strategy)



👍 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

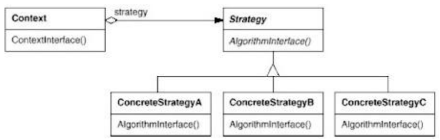
👎 Functional approach, more complex syntax needed
👎 Often considered too abstract for programmers
👎 Leverages Command objects

Batch Method

Problem: Collection and iterating client not on same machine
Solution: Group multiple collection accesses together
Implementation: Data structure for calls, Access groups of elements
Example: String Builder, SQL Cursors
Relations: Variation of (Remote Proxy)

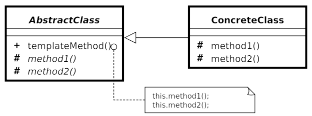
👍 Less communication overhead 👎 Increased complexity

Strategy



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 algorithms.

Template Method

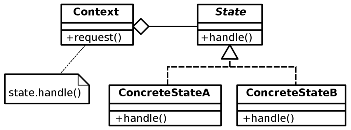


Define the skeleton of an algorithm in a operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm structure. e.g.: Document templates

State Patterns

Intent: Change Behavior based on state **Problem:** An entity behaves differently depending on its state

State (Objects for State)



- Results in a lot of classes and structures

Object acts according to its state without multipart conditional statements.

Problem: One class per state with behavior, switch state on context
Solution: Helper function to change state on context

👍 Avoid bulky conditional, Single Responsibility, Open/Closed
👎 Overkill if only few states, Behavior not in one place

Methods for State

- Propagates a single class with a lot of methods

👍 Allows a class to express all of its different behaviors in ordinary methods on itself
👍 Behavior is coupled to the state machine, rather than fragmented across multiple small classes
👍 Each distinct behavior is assigned its own method
👍 No object context needs to be passed around, methods can already access the internal state of the state machine

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

Collections for State

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

🔗 Objects for State, Methods for State

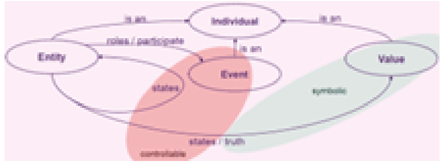
👍 No need to create a class per state
👍 Optimized for multiple objects (state machines) in a particular state
👍 Object's collection implicitly determines its state (No need to represent the state internally)

👎 Can lead to a more complex state manager

Value Patterns

System Analysis (OOA = Object Oriented Analysis)

An individual is something that can be named and reliably distinguished from other individuals.

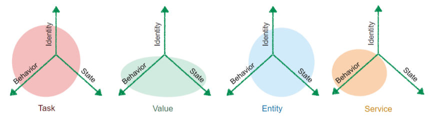


Kind of individuals:

- **Event** an individual happening, taking place at some particular point in time.
- **Entity** an individual that persists over time and can change its properties and states from one point in time to another. Some entities may initiate events; some may cause spontaneous changes to their own states; some may be passive
- **Value** an intangible individual that exists outside time and space, and is not subject to change.

Software Design (OOD)

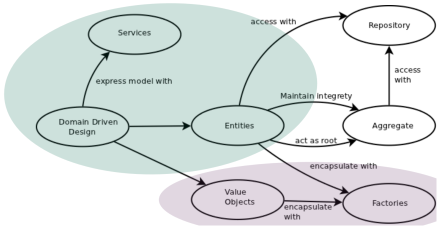
- **Identity:** significant, or transparent (=transient identity)
- **State:** object stateful or stateless
- **Behavior:** object has significant behavior independent of its state



Categories of objects:

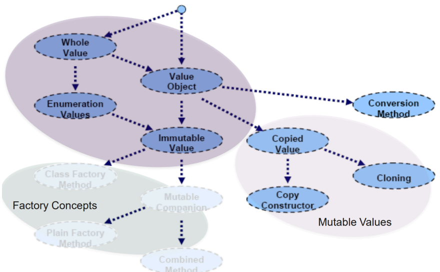
- **Entity:** Express system information (persistent). Distinguished by identity.
- **Service:** Represents system activities. Distinguished by behavior.
- **Value:** Content dominant characteristic. No significant enduring identity (transparent).
- **Task:** Represent system activities. Distinguished by identity and state (e.g. command objects, threads)

Domain Driven Design



Patterns of Value

Patterns of Value addressing value objects in "pure" OO languages.



Whole Value

Intent: Express the type of the quantity as a Value Class
Problem: Primitive quantities have no domain meaning
Solution: Recover meaning loss by providing dimension and range
Implementation: Disallow inheritance to avoid slicingValue Object
Example: `public final class Value{public Value(v){check range}}`

Value Object (Value Class)

Intent: Make Comparison based on the value
Problem: Values by default get compared by the "identity"
Solution: Override methods in object that should rely on content
Implementation: Override equality methods e.g. `equals`, `hashCode`, implement Serializable

Conversion Method

Intent: Convert from one (ctr:more generic) Value Object to another
Problem: Related value objects should be converted to each other
Solution: Constructor for Conversion, Conversion instance method `toOtherType()`) or Class Factory Method `fromOtherType()`)

Immutable Value

Intent: No side-effects when sharing or aliasing Value Objects
Problem: Sharing of instances not corresponding to value definition
Solution: Set internals at construction and allow for no modification
Implementation: declare all field private final, Mark class as final
Example: `final class D{ private final Y y; }`

Enumeration Values

Intent: Represent a fixed set of constant values and preserve type safety.
Problem: Fixed range of values should be typed (e.g. months)
Solution: Whole Value and declare **enum** values as public read only

Copied Value and Cloning

Intent: Values should be modifiable without changing origins state
Problem: Values by Reference modify internal state
Solution: Create Copy when sharing so that internal state stays intact
Implementation: For deep cloning: non-final fields, reassigned clone. `implements Cloneable`, override `clone()`
Relations: Setting context for (Prototype)

Copy Constructor

Intent: Copying without a clone method
Problem: Within Value Objects what to copy is clear→need nc `clone()`
Solution: Copy constructor consuming instance of same type
Implementation: final class, no inheritance, create copy constructor

Class Factory Method

Intent: Simplification and optimization of construction of Value Objects
Problem: Construction of Value object sometimes expensive
Solution: Use static methods instead of ordinary constructor
Example: `public final class Value {public static Value of (int v)}`
Relations: ariation of (Flyweight)

Mutable Companion

Intent: Simplify complex construction of an immutable Value
Problem: Mutation of immutable object costly or impossible
Solution: Implement Companion class that supports modifier methods
Implementation: Factory method on companion for modifications
Relations: Setting Context for(Builder), Variation(Plain Factory Method)

Relative Values

Intent: Comparison of value object based on their **state** instead of **identity**
Problem: Comparison of value objects per default by identity
Solution: Override/Implement Comparison and equality methods
Implementation: Provide Bridge method (see value object example)

CHECKS Patterns

Separate good input from bad, (validation)

Meaningful quantities

Exceptional Behavior

Intent: Handle exceptional circumstances (missing/incorrect)
Problem: Missing or incorrect values impossible to avoid
Solution: Distinguished values for exceptional circumstances

Meaningless Behavior

Intent: Handle (missing/incorrect) data without overhead
Problem: How Exceptional behavior handled without throwing errors?
Solution: Write methods with minimalistic concern for possible failure

Meta Patterns (Reflection)

👍 Adapting software system is easy, Support many changes
👎 Non-transparent "black magic" APIs, typesafety, Efficiency **Dangers:** Overengineering, "Security" undermined, obscure API, config

Type Object

Intent: Keep common behavior and data in only one place
Problem: Need to identify instances, behavior depends on their Type
Solution: Categorize objects by another object instead of a class
Implementation: Delegate calls to type object, change type -> keep id
Relations: Setting context by(Strategy), Variation of (State)

👍 Extensable categories, avoids explosion, multiple meta levels
👎 Mess of classes, lower efficiency, database schema change

Property List

Intent: Make Attributes attachable / detachable after compilation
Problem: (At/De)tachable attributes shared across the class hierarchy
Solution: Provide objects with a property list that contains attributes
👍 Extensibility, attribute iteration, attributes across hierarchies
👎 Typesafety, unchecked nameing, runtime overhead, access

Bridge Method

=> Mitigates liabilities of Property List **Intent:** Provide consistent naming and type safety to property list
Problem: Inconsistent naming and no type safety on property list
Solution: Bridge Methods with fixed name and return type

Anything

Intent: Data should be structured recursively
Problem: (At/De)tachable attributes with structure and recursion
Solution: Implement representation for simple/complex values
Relations: Specialization of (Composite),Setting Context by (Property List), Setting Context by (Null Object)
👍 Streamable format,flexible interchange across boundaries
👎 Typesafety, unapparent intent, lookup overhead

When to usee what?

Reflection

- Deep introspection and modification capabilities are needed
- Building frameworks or libraries that require dynamic behavior based on user-defined classes.
- The drawbacks are acceptable. • Large number of similar obj. that differ in characteristics, not their behaviors.

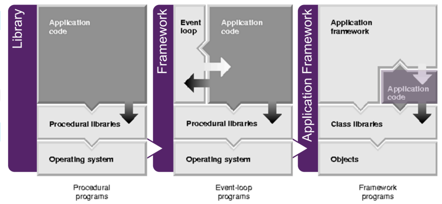
Type Object

- When avoid extensive inheritance hierarchy for different types.
- Types are known at compile time, and properties don't change dynamically so new class is required.

Property List

- Flexibility in object's attributes needed, with ability CRUD properties at runtime.
- Type safety is not a primary concern or mechanisms to handle it effectively exists.
- Dealing with scenarios where objects can be very diverse and unpredictable, and you want to avoid rigid class structures.

Frameworks



Object-oriented classes that work together, privides hooks for extension, keeps control flow. **e.g.** Hibernate, Velocity, .NET Core & EF, React.js, Vue.js, MFC

Application Framework

Object-oriented class library, `Main()` lives in the Application Framework, provide hooks to extend and callbacks, provides ready-made classes for use. **e.g.** Spring, J2EE, ASP.NET, Angular, Office, Apache, httpd, quote

Micro Frameworks

Represented by many Design Patterns **e.g.** Template Method, Strategy, Command Processor

Meta Framework

Framework adaption concerns: Acquisition cost, Long-term effect, Training and support, future technology plans, Competitor responsible
Key Ideas: Difference to other technologies, different usage contexts.

Frameworkers Dilemma

Usage: Framework users implement application code by **Portability:** Code strongly coupled to overlying Framework **Testability:** Hard to take single piece/package and write automated unit tests for application classes using Framework **Evolution:** Framework users implementation might break in next version => Check out **functional** and **non-functional** requirements and **evaluate** Framework with care, **before** get locked-in.

Ways out of dilemma: 1. Think very hard up-front, 2. Don't care too much bout framework users, 3. Let framework users participate, 4. Use helping technology