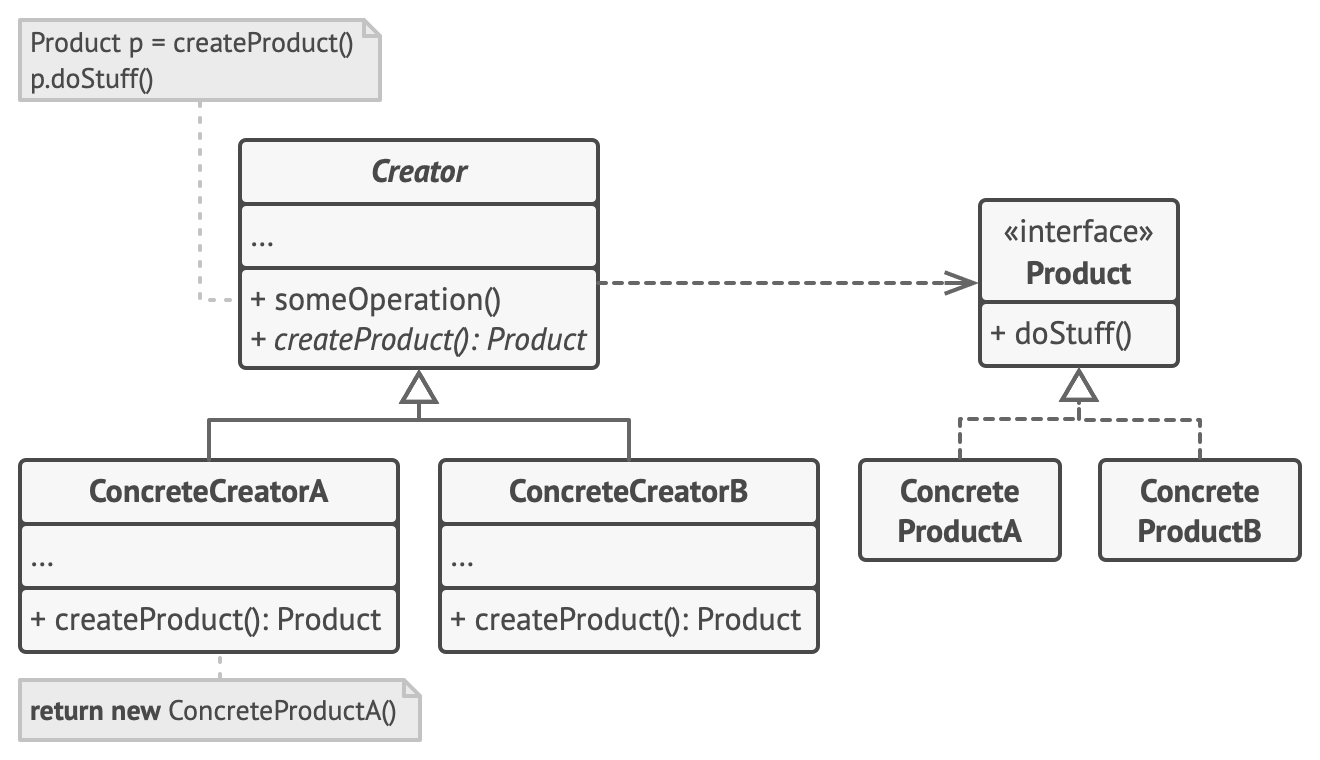
**Pattern Definition:** Descriptions of successful engineering stories with a well-known intent. Address recurring problems, nothing new. Describe generic solutions that worked. Tell about the forces of the problem. What makes the problem addressed hard? Tell about the engineering trade-offs to take. Benefits and Liabilities

Give us names to talk with and about. How to get the problem solved (process for implementation)

# creational

**Factory Method:**

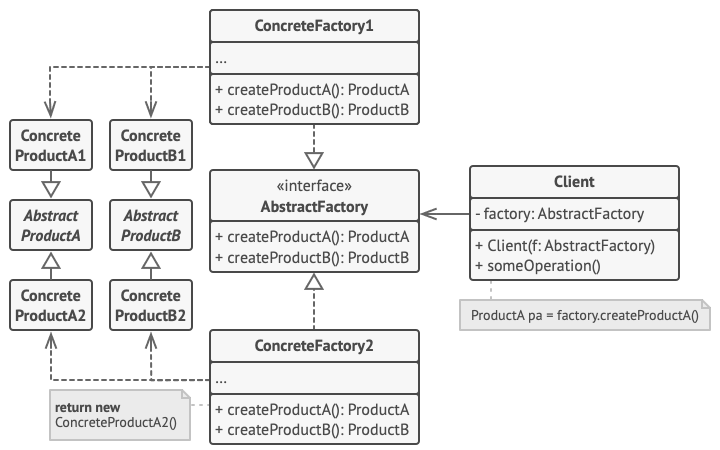
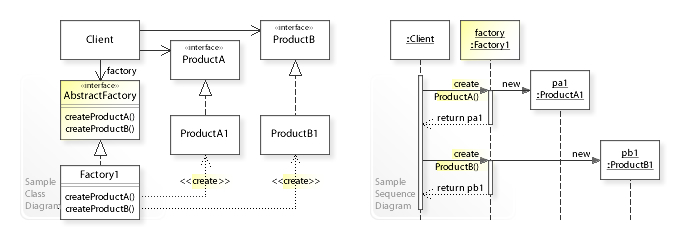
|  |  |
| --- | --- |
| Intent | Define common structure but allow subclasses to define their own logic. Provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. |
| Benefits | * Avoid tight coupling between creator and concrete products. * Single Responsibility Principle (creation code in 1 place, making it easier to support) |
| Liabilities | Code may become more complicated since there are a lot of new subclasses to implement the pattern |



**Relations:** Can evolve toward Abstract Factory, Prototype or Builder. Abstract Factory often based on a set of Factory methods.

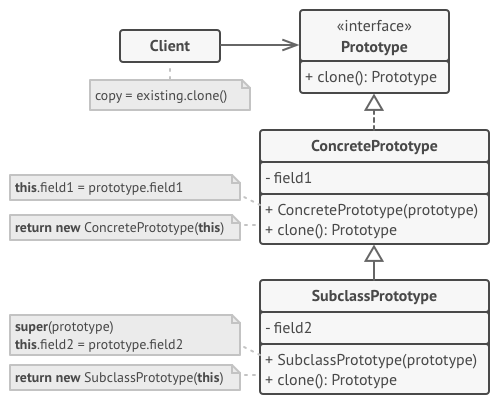
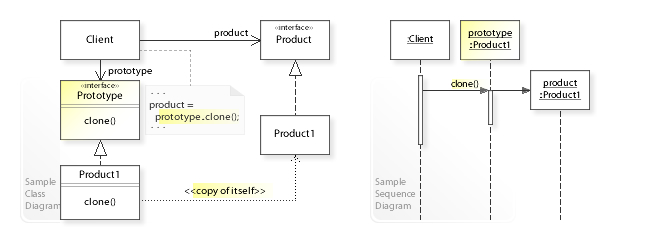
**Abstract Factory:**

|  |  |
| --- | --- |
| Intent | Produce families of related objects without specifying their concrete classes. |
| Benefits | * Products you’re getting from a factory are compatible with each other * Avoid tight coupling between concrete products and client code |
| Liabilities | The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern. |

****

**Prototype:**

|  |  |
| --- | --- |
| Intent | Application "hard wires" the class of object to create in each "new" expression. Copy existing objects without making your code dependent on their classes. |
| Benefits | * Clone objects without coupling to their concrete classes * Get rid of repeated initialization code in favor of cloning pre-built prototypes. * Alternative to inheritance when dealing with configuration presets |
| Liabilities | Cloning objects with circular references might be very tricky |

****

## singleton

**Singleton:**

|  |  |
| --- | --- |
| Intent | Some classes should have only one instance. How can be guaranteed that only one object of a class is instantiated and can globally accessed? «Ensure a class only has one instance and provide a global point of access to it.» [GOF] |
| Benefits | * Controlled access to sole instance * Reduced name space * Permits refinement of operations and representation |
| Liabilities | * Introduces a global variable/state (tight coupling, problematic with multi-threading) * Prevents polymorphism (limits interchangeability, limits unit testing (mocking) and disallows parallelism during tests, limits application evolution) * Carries state until the app closes |

**Registry:** Lookup an object knowing only its name. This pattern stores instances of objects internally and uses a dictionary mapping to retrieve instances later.

|  |  |
| --- | --- |
| Intent | Provide a centralized point for storing and retrieving objects, typically to manage global resources or to enable communication between objects in a system. It allows objects to be registered with a central registry, and other objects can then access or retrieve these objects through the registry. |
| Benefits | * Can help in decoupling components. Instead of referencing each other, they interact through the registry * Objects can be dynamically added, removed in the registry at runtime |
| Liabilities | Might introduce implicit dependency between object and registry |

|  |  |  |
| --- | --- | --- |
| **Singleton** |  |  |
| **static Instance()** |  | **return Lookup(“DEFAULT”);** |
| **SingletonOperation()** |  |
| **GetSingletonData()** |  |
| **Register(name, singleton)** |  |  |
| **Lookup(name)** |  |  |
| **static uniqueInstance** |  |  |
| **singletonData** |  |  |

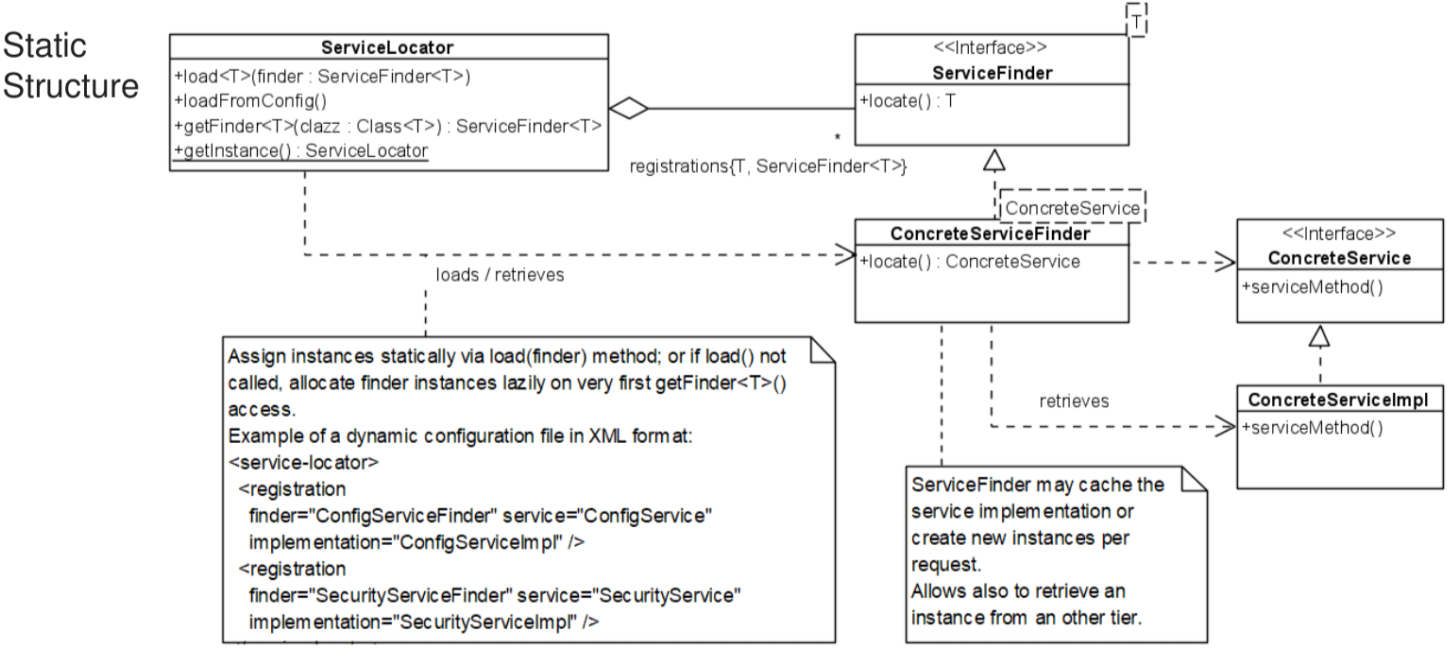
**Monostate:**

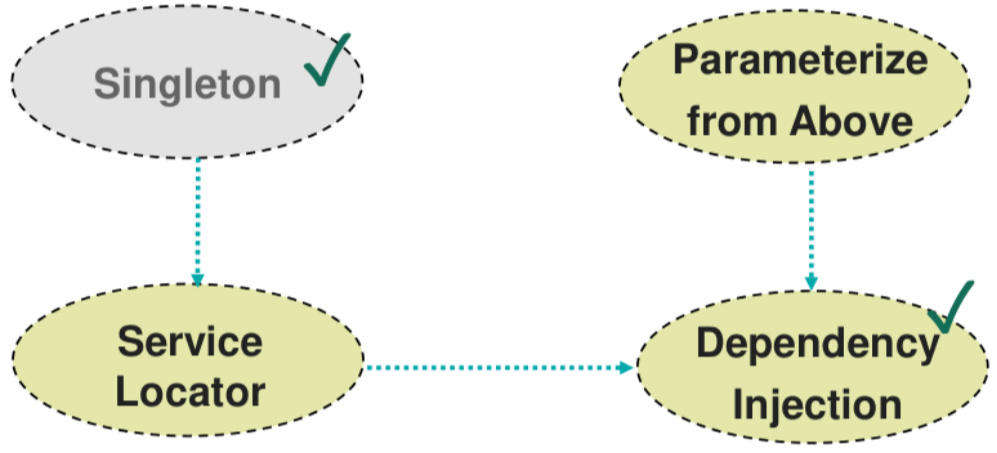
|  |  |
| --- | --- |
| Intent | Make all instances of a class share a single, common state, essentially making them appear as if they were instances of the same object. |
| Benefits | All instances of the class have the same state, ensuring uniformity across the objects |
| Liabilities | Might be counterintuitive for developers who are not aware of the pattern. Changes to the state of one instance affect all instances, which can lead to surprising behavior. |

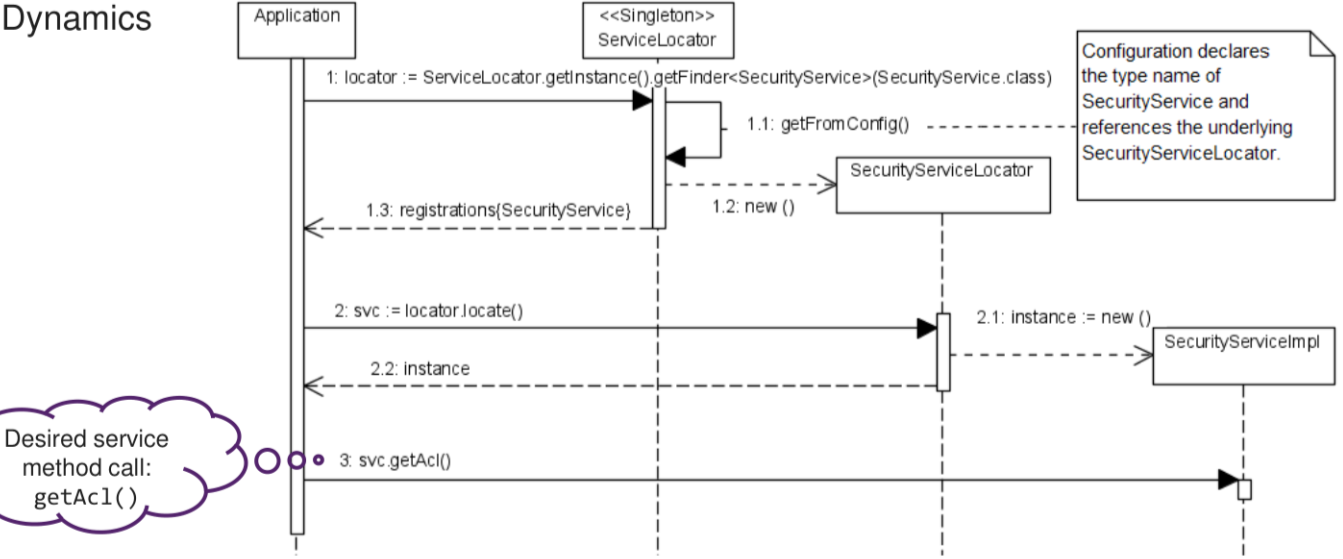
public interface Monostate { int getY(); }  
public class MonostateImpl implements Monostate {  
 public int getY() {  
 return Singleton.getInstance().getY();  
 }  
}  
public class MonostateMockImpl implements Monostate {  
 private static MockSingleton = new MockSingleton();  
 public int getY() { return MockSingleton.getY(); }  
}

**Service Locator:**

|  |  |
| --- | --- |
| Intent | Instead of every service having their own singleton, there is only 1 singleton, which can be used for all services |
| Benefits | * Only a single singleton is used * All others are polymorphic with interfaces * Services can be exchanged even at runtime |
| Liabilities | Clients still rely on a static reference to ServiceLocator (tight coupling) hence ServiceLocator can’t be removed |

****

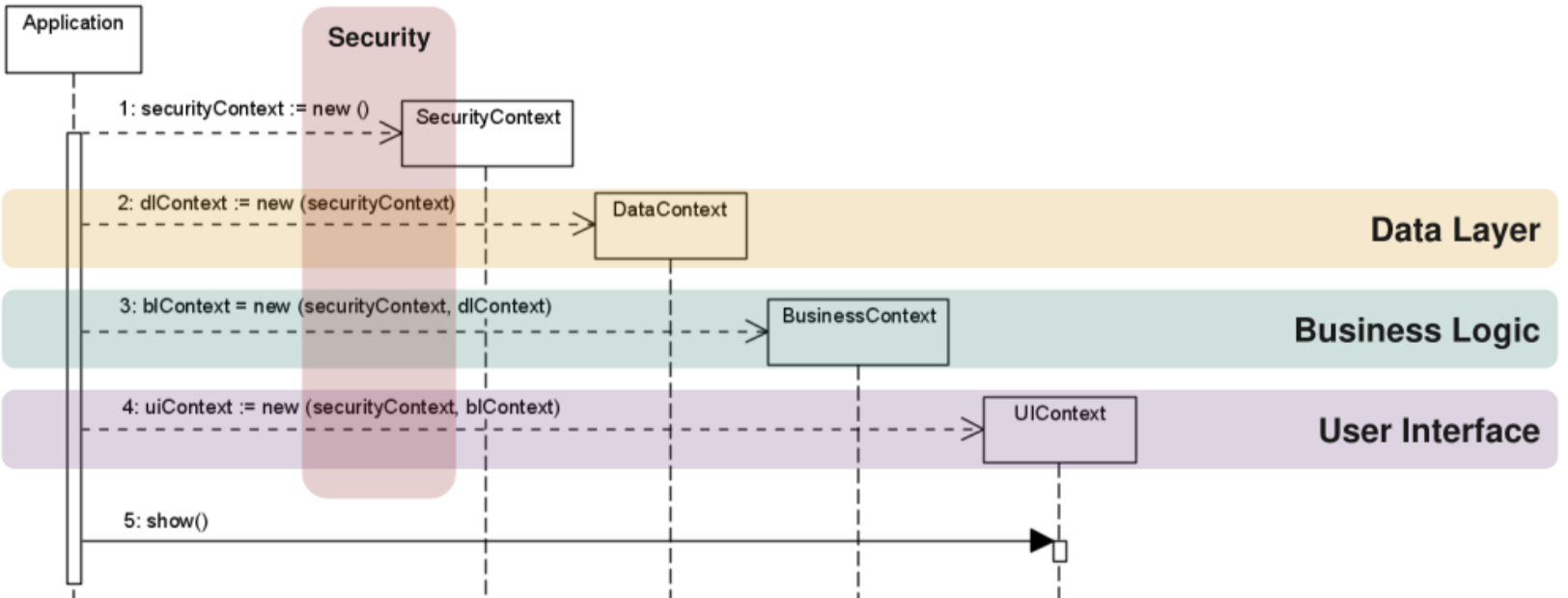
****

****

**Parameterize from Above:**

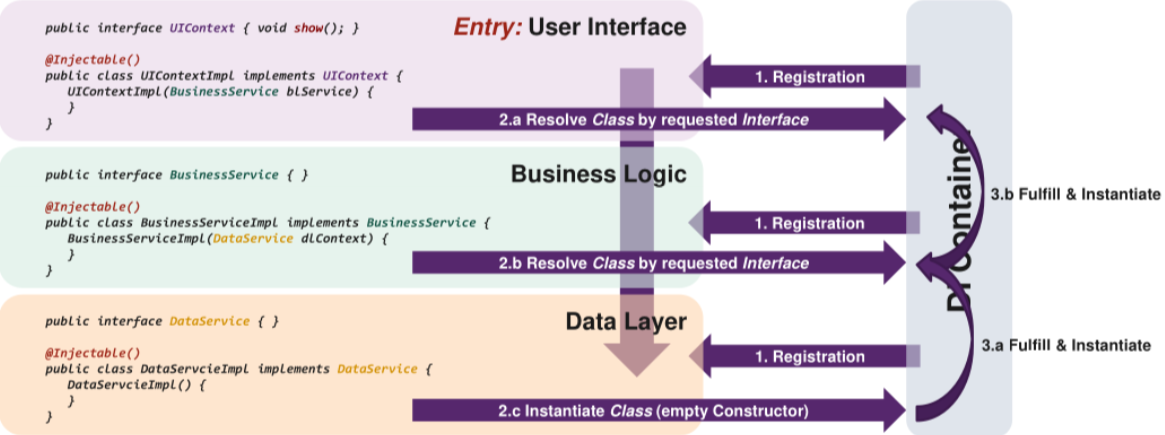
|  |  |
| --- | --- |
| Intent | How can I provide individual services that need to run independently of each other with global data without using singletons? Define global services and put them into the application, then register “horizontal” services one by one |
| Benefits | * No global variables * Implementations of parametrized functionalities are exchangeable * Enforces separation of concerns at architecture level (view, logic and data are separated) |
| Liabilities | * Complexity because object instances aren’t accessible from everywhere; access to application context needed * Contexts must be passed through the entire application stack * Fragile bootstrapper: application must be wired completely at startup |

public final class Bootstrapper {  
 public static void main(string[] args) { *// PfA applied  
 // instantiate vertical layer contexts first* SecurityContext securityContext = new SecurityContextImpl();  
 ConfigurationSettings configuration = new ConfigurationSettingsImpl(args);  
 *// encapsulate variables into an application context* var applContext = new ApplicationContextImpl(  
 securityContext, configuration);  
 *// instantiate horizontal layer contexts from bottom to top* DataContext dlContext = new (applicationContext);  
 BusinessContext blContext = new BusinessContextImpl(applContext, dlContext);  
 UIContext uiContext = new UIContextImpl(applicationContext, blContext);  
 *// show initial UI dialog* uiContext.show();  
 }  
}

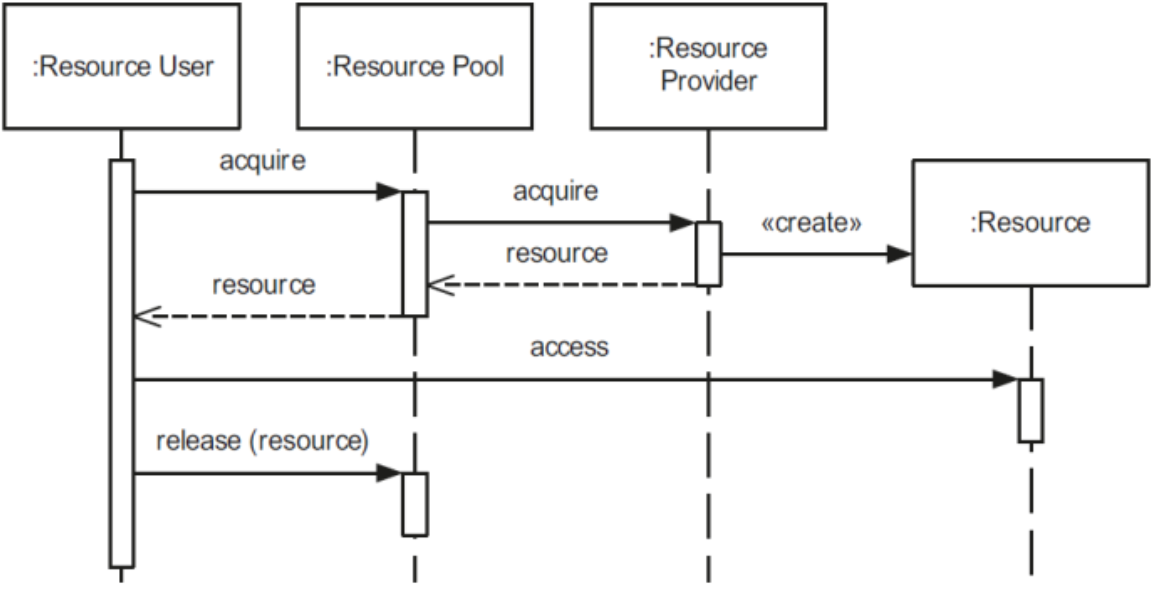


**Dependency Injection:**

|  |  |
| --- | --- |
| Intent | Reduce the coupling between classes by externalizing the creation and management of dependencies. |
| Benefits | * Decoupling * More flexibility in configuring and replacing components |
| Liabilities | Harder to understand (seems like magic) |

**  
Pooling:**

|  |  |
| --- | --- |
| Intent | I require fast/efficient access to resources that need to be available to multiple objects. Create a pool of resources which can be acquired or released by clients. |
| Benefits | * Improves the performance of an application by reducing the time spent in costly release and re-acquisition * Lookup and release of previously-acquired resources is predictable * Simplified release and acquisition of resources |
| Liabilities | * The management of resources results in a certain overhead * Depending on the environment and resource type, resources must be released back to the pool * Acquisition requests must be synchronized to avoid race conditions |

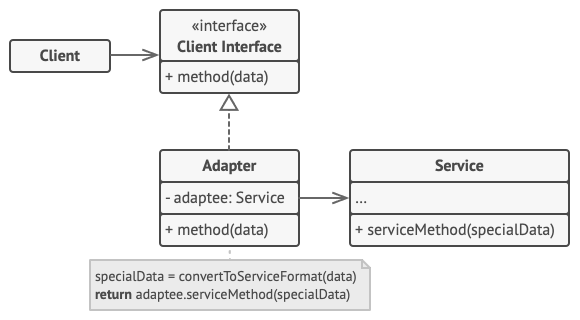
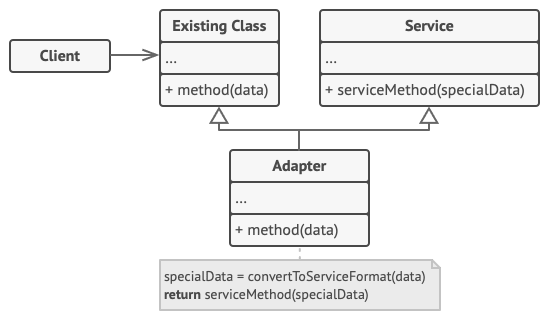
****

# structural

**Adapter:**

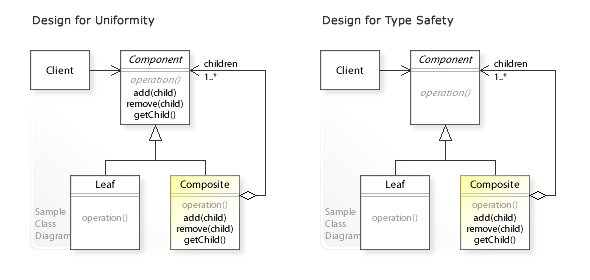
|  |  |
| --- | --- |
| Intent | How to implement a reusable “off the shelf” functionality. Allows objects with incompatible interfaces to collaborate. E.g convert XML to JSON. |
| Benefits | * Separate the interface or data conversion code from the primary business logic * Introduce new types of adapters into the program without breaking the existing code |
| Liabilities | Overall complexity of the code increases. Sometimes it’s simpler to change the service class to match the rest of the code |

Object adapter Class adapter

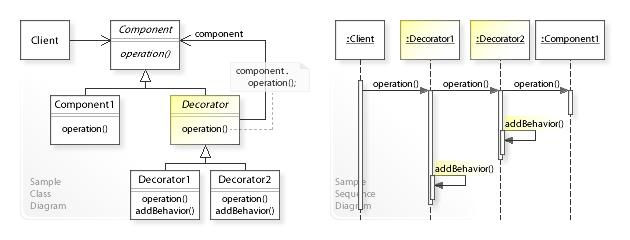
**Composite:**

|  |  |
| --- | --- |
| Intent | Application needs to manipulate a hierarchical collection of objects. Compose objects into tree structures and then work with these structures as if they were individual objects. E.g box that either contains more boxes or a product. |
| Benefits | Work with complex tree structures more conveniently with polymorphism and recursion |
| Liabilities | Might be difficult to provide common interface for classes whose functionality differ too much. In certain scenarios, you’d need to overgeneralise the component interface, making it harder to understand |

****

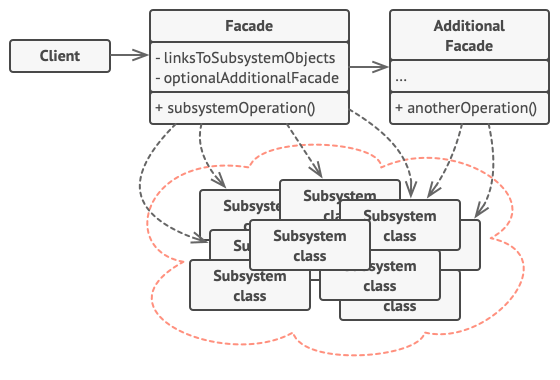
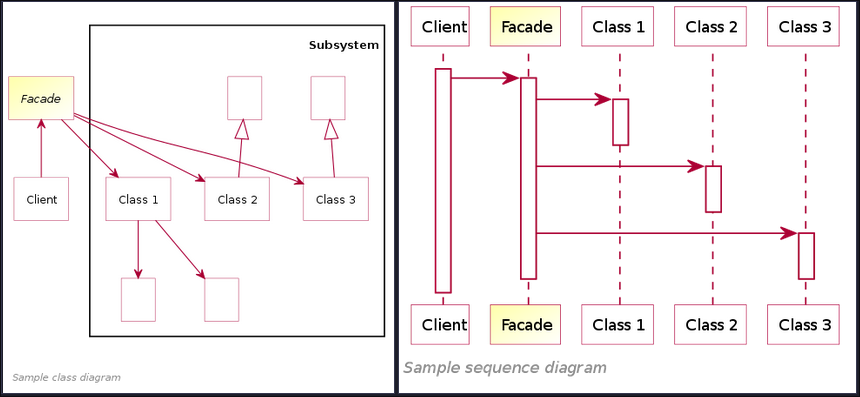
**Decorator:**

|  |  |
| --- | --- |
| Intent | Add behavior or state to individual objects at run-time. Inheritance is not feasible because it’s static and applies to an entire class. Attach new behaviours to objects by placing these objects inside special wrapper objects that contain the behaviours. |
| Benefits | * Extend an object’s behavior without making a new subclass * Add/Remove responsibilities from object at runtime * Combine several behaviours by wrapping an object into multiple decorators |
| Liabilities | * Hard to remove a specific wrapper from the wrappers stack * Hard to implement decorator so that its behaviour doesn’t depend of the order in decorator stack |

****

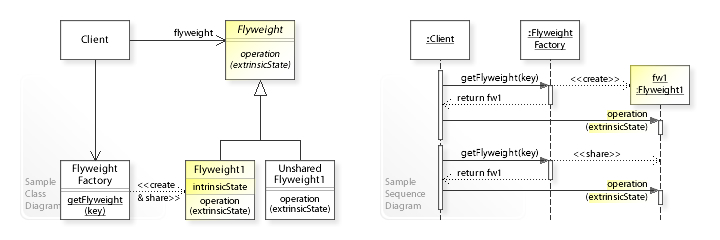
**Facade:**

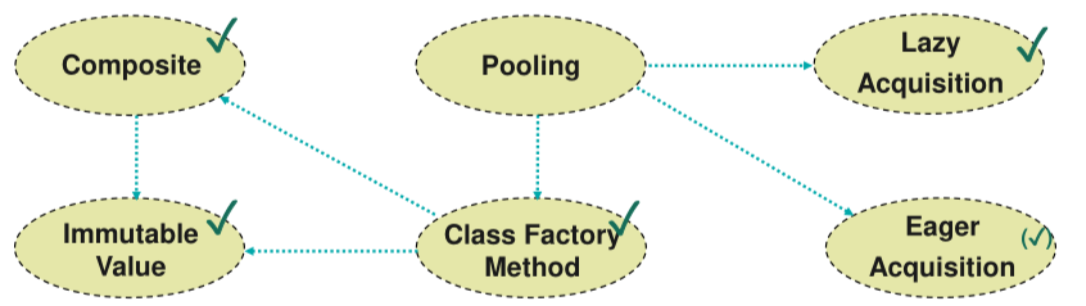
|  |  |
| --- | --- |
| Intent | Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. |
| Benefits | Isolate your code from the complexity of a subsystem. |
| Liabilities | Can become a god object (object that references many distinct types, has too many unrelated/uncategorized methods) coupled to all classes of an app. |

****

**Flyweight:**

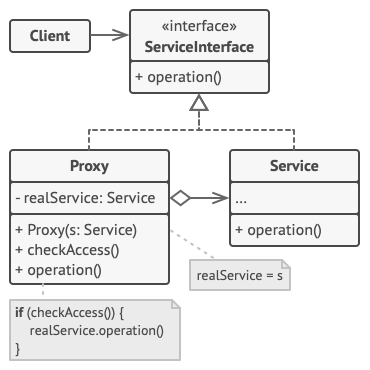
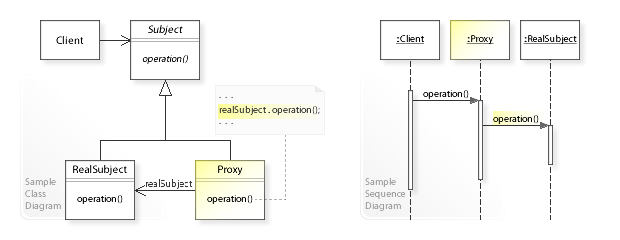
|  |  |
| --- | --- |
| Intent | Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage. Sharing common parts of state between multiple objects instead of keeping all of the data in each object. |
| Benefits | Save lots of RAM, assuming your program has tons of similar objects. |
| Liabilities | * Might be trading RAM usage for CPU usage, when some of context data needs to be recalculated each time somebody calls a flyweight method * Code becomes much more complex |

****

****

**Proxy:**

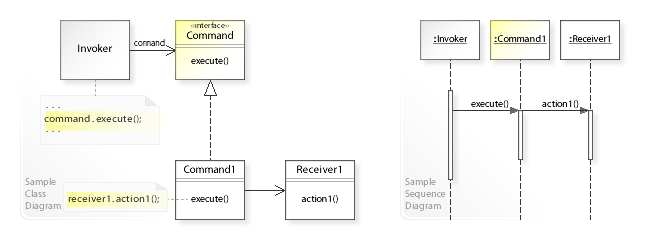
|  |  |
| --- | --- |
| Intent | Need to support resource-hungry objects, but do not want to instantiate such objects unless and until they are requested by the client. Provide a placeholder for another object to control access to it. (Db) |
| Benefits | * Control service objects without clients knowing about it * Manage lifecycle of service object * Proxy works even if service is not ready or not available |
| Liabilities | * More complicated because of a lot of new classes * Response from the service might get delayed. |

****

# behavioural

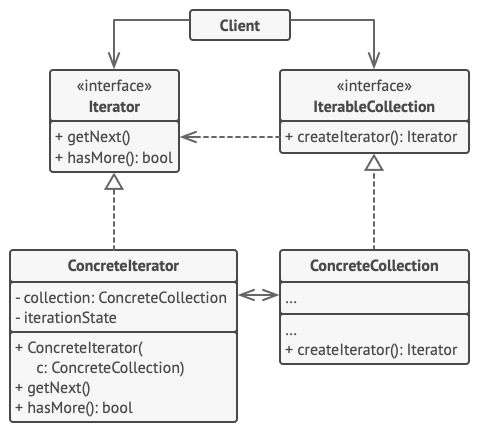
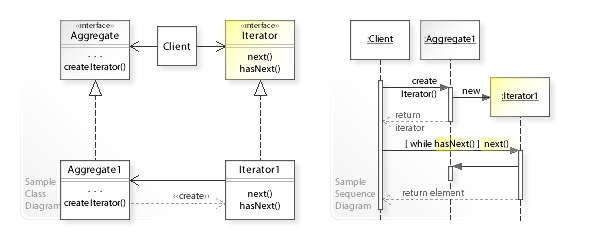
**Command:**

|  |  |
| --- | --- |
| Intent | Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request’s execution. |
| Benefits | * Decouple classes that invoke operations from classes that perform operations * Implement deferred execution of operations. |
| Liabilities | May become more complicated since there is a whole new layer between senders and receivers. |

****

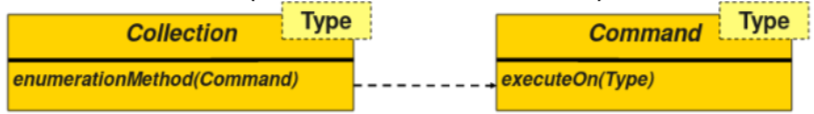
**External Iterator:**

|  |  |
| --- | --- |
| Intent | Need to "abstract" the traversal of different data structures so that algorithms can be defined that are capable of interfacing with each transparently. Traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.). |
| Benefits | * Iterate over the same collection in parallel because each iterator object contains its own iteration state * Clean up the client code and the by extracting bulky traversal algorithms into separate classes |
| Liabilities | * Overkill if app only works with simple collections * May be less efficient than going through elements of collection directly |

****

**Internal Iterator:**

|  |  |
| --- | --- |
| Benefits | No loop housekeeping, Synchronization at whole traversal |
| Liabilities | More complex syntax, needs command objects, too abstract |

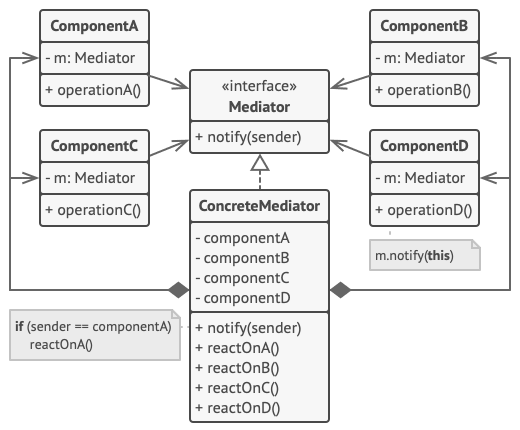
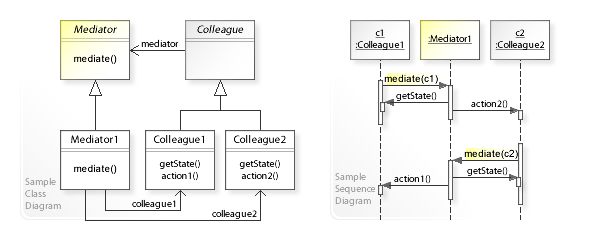
****

**Batch method Iterator:**

|  |  |
| --- | --- |
| Intent | Collection and iterating client not on same machine. Iterator would not be efficient over network. Solution is to group multiple collection accesses together (String Builder, SQL Cursors) |
| Benefits | Less communication overhead |
| Liabilities | Increased complexity |

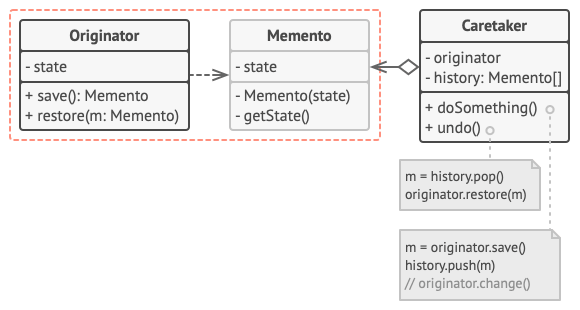
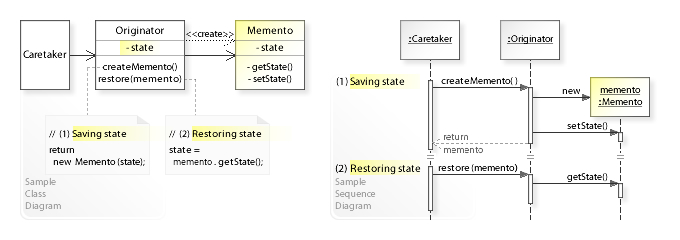
**Mediator:**

|  |  |
| --- | --- |
| Intent | We want to design reusable components, but dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon. Reduce dependencies between objects. Restricts direct communications between objects and forces them to collaborate only via a mediator object. |
| Benefits | * Reduce coupling between various components of a program. * Reuse individual components more easily. |
| Liabilities | Over time a mediator can evolve into a God Object. |

****

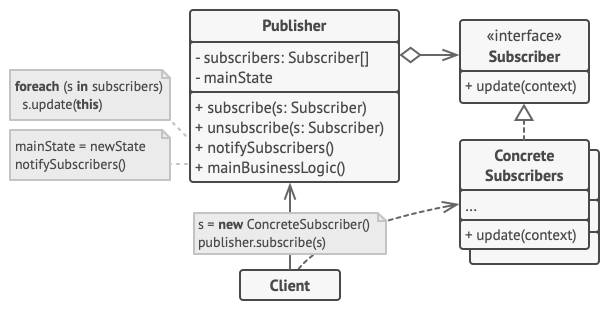
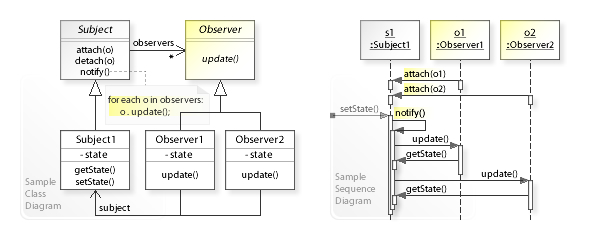
**Memento:**

|  |  |
| --- | --- |
| Intent | Need to restore an object back to its previous state ("undo" operations). Save and restore the previous state of an object without revealing the details of its implementation. Delegates creating the state snapshots to the originator (owner of state). |
| Benefits | * Produce snapshots of the object’s state without violating its encapsulation. * Simplify the code by letting the caretaker maintain the history of the originator’s state |
| Liabilities | * Might consume lots of RAM if clients create mementos too often * Caretakers should track the originator’s lifecycle to be able to destroy obsolete mementos. |

****

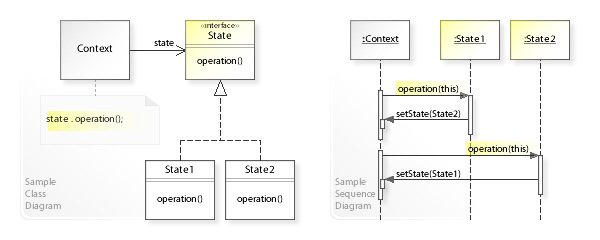
**Observer:**

|  |  |
| --- | --- |
| Intent | A large monolithic design does not scale well as new graphing or monitoring requirements are levied. Define a subscription mechanism to notify multiple objects about any events that happen to the object they’re observing. |
| Benefits | Establish relations between objects at runtime. |
| Liabilities | Subscribers are notified in random order |

****

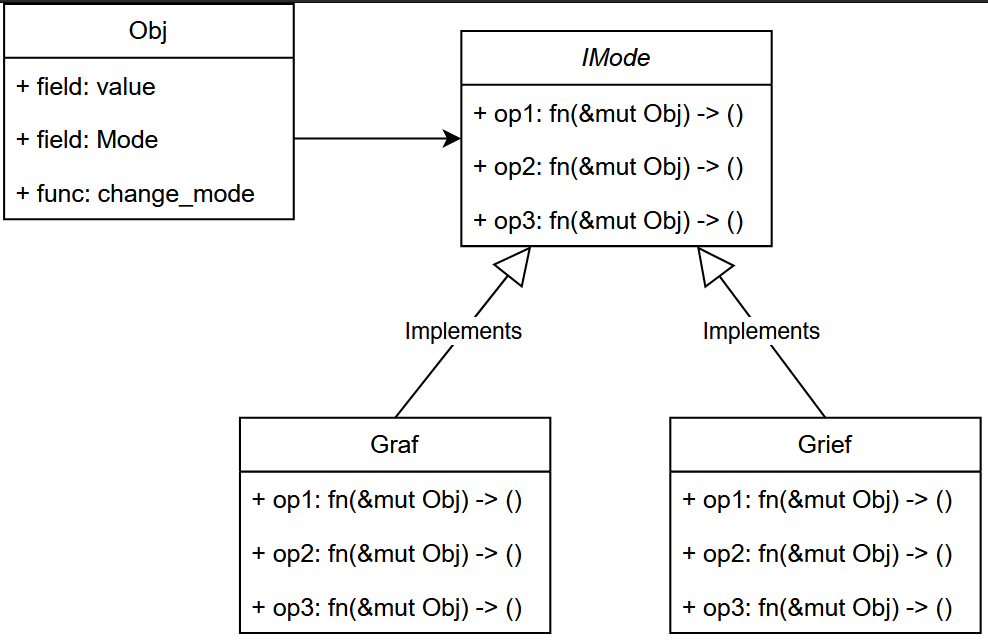
**State (for objects):**

|  |  |
| --- | --- |
| Intent | Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class. wrapper + polymorphic wrappee + collaboration |
| Benefits | * Simplify the code of the context by eliminating bulky state machine conditionals * Organize the code related to particular states into separate classes. |
| Liabilities | Applying the pattern can be overkill if a state machine has only a few states or rarely changes. |

****

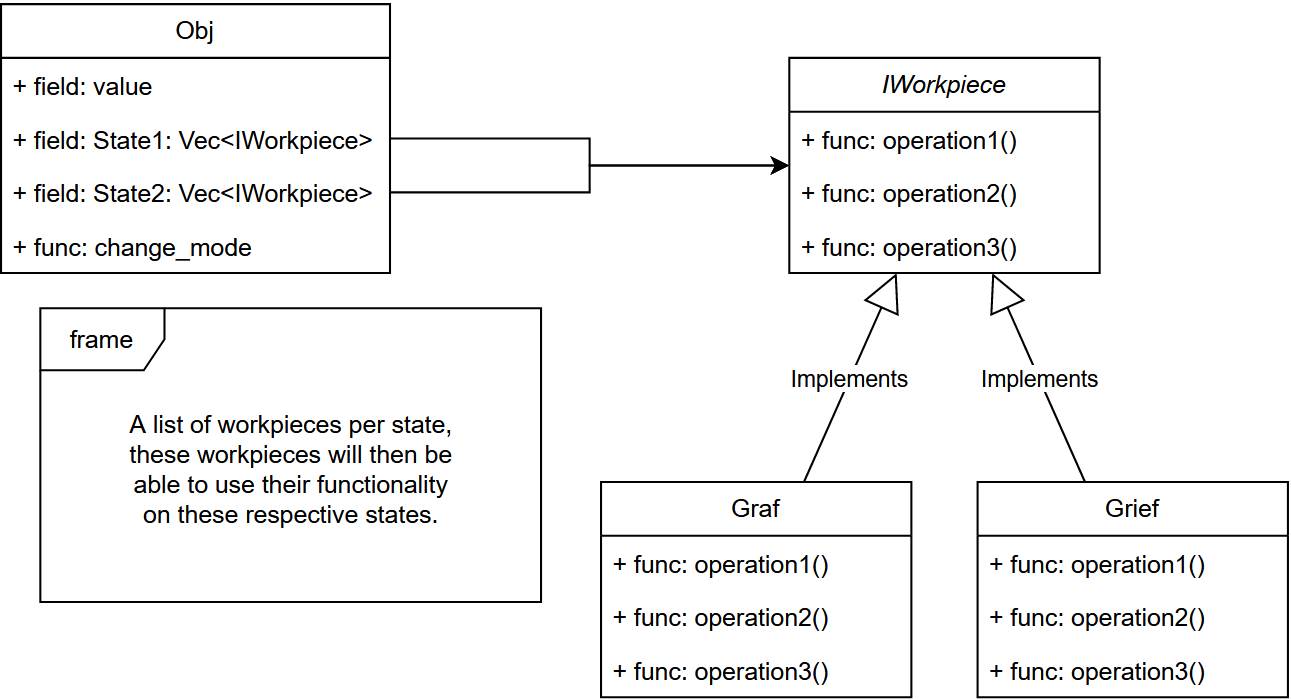
**State (methods for states):**

|  |  |
| --- | --- |
| Intent | While the object state pattern does solve the underlying problem, it also creates indirection problems which means a complicated structure that is hard to change later. In order to avoid this, we can use function pointers instead. |
| Benefits | * Slightly better performance due to removed typecasting -> virtual objects * Behavior is coupled to state machine and not into thousands of classes |
| Liabilities | Indirection still exists -> performance loss is still there. List of methods might get too large to properly manage |

****

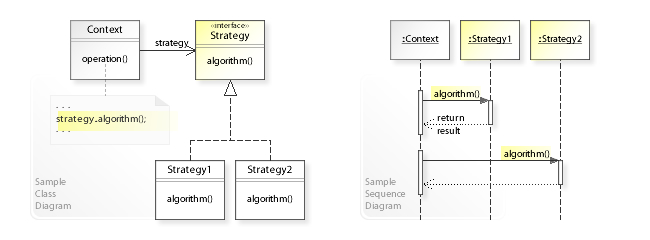
**State: (collection for states):**

|  |  |
| --- | --- |
| Intent | In order to solve the problem of both too many objects, but also too many functions inside one object, we create multiple different state machines, e.g. one for every state. Each state then uses a “Mode” class, here called workpiece that will take the function of  the state and apply it via lambda. |
| Benefits | * No need to create a class per state * Optimized for multiple objects (state machines) in a particular state. * Can be combined with other State Machine (Objects / Methods) approaches |
| Liabilities | Can lead to a more complex state manager |

****

**Strategy:**

|  |  |
| --- | --- |
| Intent | Define a family of algorithms, put each of them into a separate class, and make their objects interchangeable. E.g navigation app with different strategies to get from point A to B. |
| Benefits | * Swap algorithms used inside an object at runtime. * Isolate the implementation details of an algorithm from the code that uses it. * Replace inheritance with composition |
| Liabilities | * Don't overcomplicate with unnecessary classes/interfaces for a few algorithms * Clients must be aware of the differences between strategies to be able to select a proper one. |

****

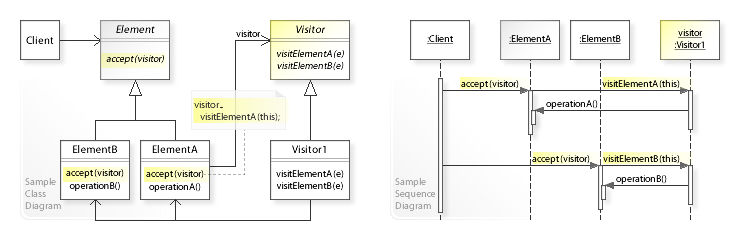
**Template Method:**

|  |  |
| --- | --- |
| Intent | Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended. Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure. |
| Benefits | * Pull the duplicate code into a superclass * Override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm. |
| Liabilities | * May be limited by the provided skeleton of an algorithm * Template methods tend to be harder to maintain the more steps they have. |



**Visitor:**

|  |  |
| --- | --- |
| Intent | Separate algorithms from the objects on which they operate. |
| Benefits | * Introduce new behaviour that can work with different classes without changing these classes * Can accumulate some useful information while working with various objects. Might be handy when traversing some complex object structures, such as an object tree |
| Liabilities | * Update all visitors each time a class gets added to or removed from the element hierarchy * Visitors might lack the necessary access to the private fields and methods of the elements that they’re supposed to work with. |

****

# value pattern

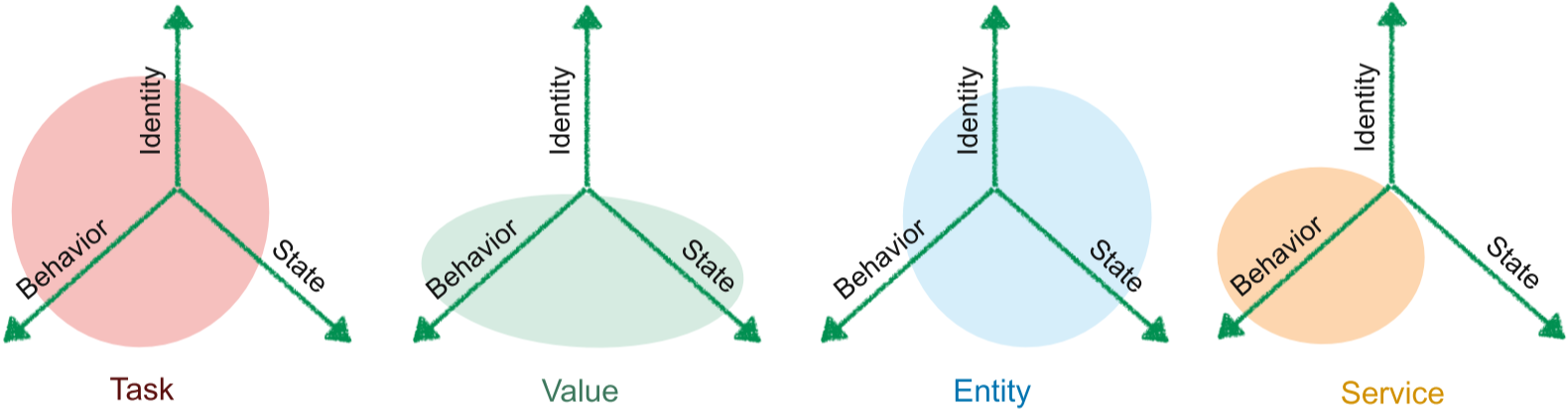
**System Analysis:** An individual is something that can be named and reliably distinguished from other individuals.

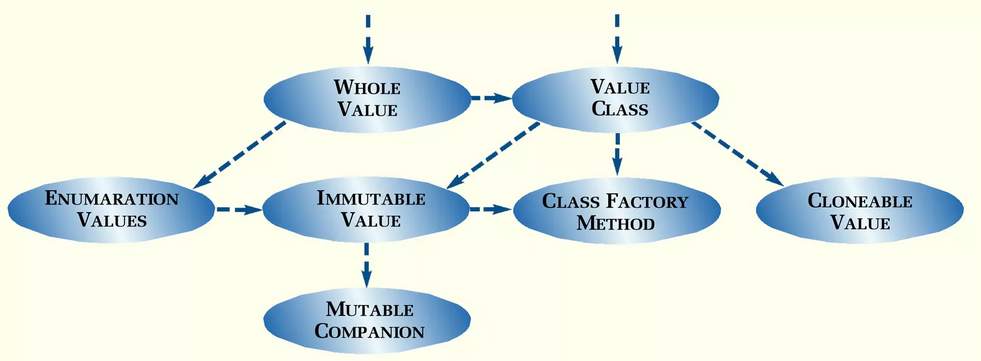
* **Events:** Individual happening, taking place at some point in time
* **Entities:** Individual that persists over time and can change its properties and states from one point in time to another. Some entities may initiate events.
* **Values:** Intangible individual that exists outside time and space and is not subject to change

**Software Design:**

* **Entity:** System information, typically of persistent nature. Identity is important in distinguishing entity objects from one another.
* **Service:** Service-based objects represent system activities. Services are distinguished by their behavior rather than their state content or identity.
* **Values:** For value-based objects interpreted content is the dominant characteristic, followed by behavior in terms of its state. In contrast to entities, values are transient and do not have significant enduring identity.
* **Task:** Like service-based objects, task-based objects represent system activities. However, they have an element of identity and state, e.g. command objects and threads.

**Object Aspects:** Identity (Identity significant, or transparent (=transient identity)), State (Object stateful or stateless?), Behaviour (Does object have significant behavior independent of its state?)



****

**Whole value:**

|  |  |
| --- | --- |
| Intent | Plain integers and floating-point numbers are not very useful as domain values. A Whole Value class recovers the loss of meaning and checking by providing a dimension and range. |
| Benefits | Enforces value is withing range of meaning |
| Liabilities | Change in meaning requires change in code -> less flexible |

|  |  |
| --- | --- |
| public class Date {  public Date(int year,  int month,  int day) {  } } | public final class Year {  private final int value;  public Year(int year) { value = year; }  public int getvalue() { return value;} } |

**Value object:**

|  |  |
| --- | --- |
| Intent | Comparison, indexing and ordering should not rely on objects identity but its content. Override Object‘s methods who define equality (equals(), hashCode(), toString(), etc) |
| Benefits | * Allows you to compare values with identity and multiple values within |
| Liabilities | * Hashing might not be simple * Hash collisions possible |

public final class Date implements Serializable {  
 *// Store date stuff* @Override  
 public int hashCode() { return Objects.*hash*(year, month, day); }  
}

**Conversion method:** Often Value Objects are somehow related but cannot be used directly without conversion. Provide a constructor which converts between types. Can be done with overloading constructur, creating a conversion instance method or a class factory method. (toInt())

**Immutable value:** A value exists outside time and space and is not subject to change. Set internal state at construction and declare all fields private final or class as final.

**Enumeration value:** A fixed range of values should be typed (e.g. months). Implement a Whole Value and declare the Enumeration Values as public read-only fields. Or just use enum keyword

public enum Month {  
 *JANUARY*(1),   
 *// …  
 DECEMBER*(12);  
 private final int value;

private Month(int value) {  
 if (value < 1 || value > 12) { *// avoid careless mistakes, check value range* throw new IllegalArgumentException();  
 }  
 this.value = value;  
 }  
  
 public int getValue() { return value; }  
}

**Copied Value and Cloning:** Values should be modifiable without changing the origins internal state. Create a copy of the current instance and all containing fields

**Copy constructor:** Within Value Objects we often know exactly what to copy. How can objects be copied without implementing a clone method? Declare the class as final and derive from Object only. Create a copy constructor, which consumes an instance with same type

|  |  |
| --- | --- |
| Order firstOrder, secondOrder; Date date = new Date(year, month, day); firstOrder.setDeliveryDate(date); date.nextYear(); | |
| public final class Date {  public void nextYear() {  year = new Year(year.getValue() + 1);  }  public **Date(Date other)** {  *// ...* this.year = new Year(other.year);  } } | class Order {  private Date deliveryDate;   public void setDeliveryDate(  Date newDeliveryDate) {  deliveryDate = **new Date**(newDeliveryDate);  }  public Date getDeliveryDate() {  return **new Date**(deliveryDate);  } } |

**Mutable companion:** We need a way to mutate immutable variables later on, but that’s impossible. However, we can make a wrapper class around the immutable object and just mutate that instead, e.g creating a new instance of the immutable object which isn’t immutable

**Relative Values:** When we compare reference objects, we compare only the reference, how can we make sure to compare the type/value instead? Create comparator, implement Eq trait(rust), overload comparison operators (C++).

# CHECKS PATTERN

**Exceptional Behaviour:**

|  |  |
| --- | --- |
| Intent | Handle exceptional circumstances (missing/incorrect). Idk method just checks if input is correct. Result |
| Benefits | Missing or incorrect values impossible to avoid |
| Liabilities | Write methods with minimalistic concern for possible failure |

export enum CalcErr {  
 *DivByZero* = "div/0",  
 *DivisorIsNaN* = "NaN(divisor)“  
}  
export class Calculator {  
 public *div*(num: number, div: number): number | CalcErr {  
 if (divisor === 0) { return CalcErr.*DivByZero*; }  
 if (*isNaN*(div)) { return CalcErr.*DivisorIsNaN*; }  
 return numerator / divisor;  
 }  
}

**Meaningless Behaviour:**

|  |  |
| --- | --- |
| Intent | Handle (missing/incorrect) data without overhead. Throws |
| Benefits | How Exceptional behavior handled without throwing errors? Program passes problem for future program to handle |
| Liabilities | Write methods with minimalistic concern for possible failure |

public static int divide(int numerator, int divisor) throws ArithmeticException {  
 return numerator / divisor; *// CAUTION: x / 0 (integer) will throw an ArithmeticException*}

# FRAMEWORKS

Object-oriented classes that work together. Many design patterns are microframeworks. Framework provides hooks for extension. Clients extends framework classes or interfaces. In contrast to a library, a framework keeps the control flow not your extension component.

**Benefits:** Less code to write, More reliable and robust code, More consistent and modular code, More focus on area of expertise, Reusable solutions

**Application Frameworks:** Application architecture founded by the framework (Main() lives in Application Framework). Provides hooks to extend, callbacks and ready-made classes for use. (Spring, ASP.NET, Angular)

**Micro Frameworks:** Many Design Patterns represent Micro Frameworks (Template Method, Strategy, Command Processor)

**Frameworkers Lock-In:** Framework users implement application code by subclassing of framework classes and configuration of predefined components as well. Inheritance is one of the strongest couplings between two components.

* **Portability:** Code is strongly coupled to the overlying Framework. Hard to separate application specific code from the Framework details. Porting code to another frameworks gets tricky. Framework often bind to a specific platform or environment
* **Testability:** Frameworks are about reuse of components, patterns and architecture. This leads often to close coupling between the framework parts. Hard to take just a single piece/package from a typical framework or write automated unit tests
* **Evolution:** Framework user's implementation might break due next Framework version or the Framework developer discontinues the product (or parts of it). Using frameworks might result in a strong coupling to the Framework developer

**Meta Frameworks:** A Framework for evaluating software technology. Initial technology acquisition cost. Long-term effect on quality, time to market, and cost of the organization’s products and services when using the technology. Training and support services’ impact of introducing the technology. Key idea behind technology evaluation framework is:

* Understanding how the evaluated technology differs from other technologies
* Understanding how these differences address the needs of specific usage contexts.

**Developing Frameworks:** Frameworks need evolutionary improvement. Stagnancy in framework development is often interpreted prematurely (dead?) There are two reasons for lack of framework evolution and improvement:

* **No application uses the framework:** No users, no need to improve or evolve, no experience on what to improve
* **One or more applications use the framework:** Changing the framework risks breaking application. Applications produce work-arounds to framework deficiencies and break when those “features” are fixes.

**Dilemma:** Potential ways out of the frameworkers dilemma

* **Think very hard up-front:** Requires very experienced persons. Without concrete applications, it is hard to decide what to abstract and generalize in a framework. Can lead to expensive and unusable over-engineered frameworks. Takes too long to hit window of opportunity.
* **Don't care too much about framework users:** Lay the burden of porting applications to the application's developer. Provide many good and useful new features to make porting a "must". Might require porting tools, training/guidelines and conventions
* **Let framework users participate:** Social process can help, e.g. by giving users time to migrate. Deprecated interfaces that never gets deleted. Tendency for infinite backward compatibility. Design by committee is almost always over-engineered and brittle
* **Use helping technology:** Less direct code-dependencies. Simple and flexible interfaces (tendency to be more stable). Use configuration to reduce code-dependencies. Let the framework do as much as possible without writing code

# META PATTERN

Reflection often used as technology for Meta Programming. Most frameworks use Reflection to locate implementations of extension points and hooks.

**Recurring Problems:** Exchanging parts of software system is hard. Not yet-unknown software components should be integrated.

**Definition:** The program can observe itself (Introspection, e.g query object properties, get list of methods) and modify its own execution state (Intercession, e.g modify object properties, exchange code)

**Usage:** Load Jar/DLL at runtime, invoke methods, read properties/fields, create object instances, etc. Provides facility to implement DI, Convention over Configuration, ORM

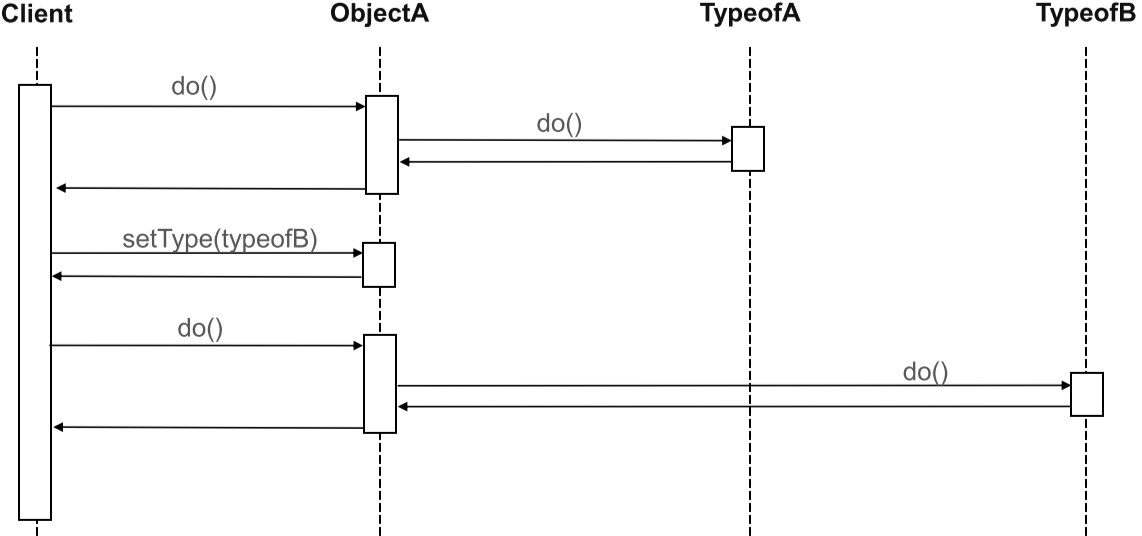
**Summary:**

|  |  |
| --- | --- |
| Benefits | Liabilities |
| Adapting a software system is easy | Produces non-transparent “black magic” APIs (hard to understand control flow) |
| Support for many kinds of changes | Binding at runtime (Late Binding) (Limited type safety, no compiler optimisation -> lower efficiency) |

**Dangers:** Costs efficiency, much work to configure, overengineered solutions, security mechanisms are undermined

**Type Object:**

|  |  |
| --- | --- |
| Intent | Keep common behavior and data in only one place. Object behaviour depends on category. Create a category (type) object which  describes multiple objects. Allow objects to change their type at runtime while keeping identity. |
| Benefits | * More categories can be added easily * Avoid explosion of subclasses with attributes etc. |
| Liabilities | * Lower efficiency because of indirection * Separation not always easy to understand directly |



|  |  |
| --- | --- |
| *// Base Level*  public class Copy {  protected MediaType type; *// typeof this* protected int copyid; *// e.g. inventory no   // current identity of this copy* public int getId() { return copyid; }   *// example of delegation* public String getTypeId() { return type.getId(); }  public String getTitle() { return type.getTitle(); } } | *// Meta Level* public class MediaType {  protected String title;  protected String typeid;   public String getId() { return typeid; }  public String getTitle() { return title; } } |

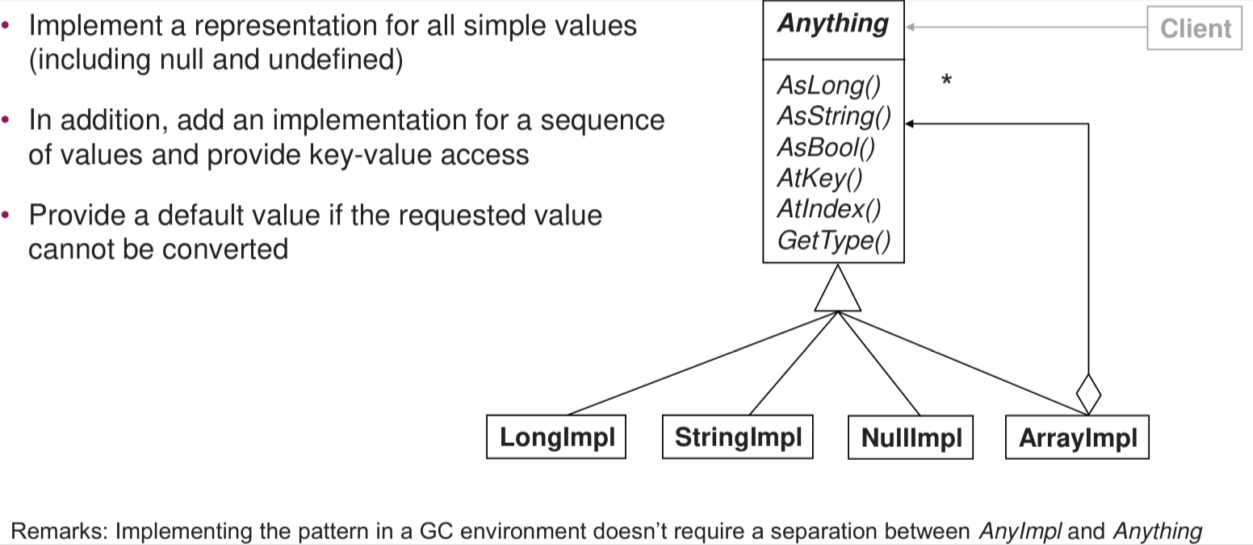
**Property List:**

|  |  |
| --- | --- |
| Intent | Attributes should be attachable / detachable after compilation. Objects share attributes / parameters across the class hierarchy. Solution is to map attribute names to values in a property list. |
| Benefits | * Black box expendability * Object extension while keeping identity and type |
| Liabilities | * Indirection for each property * No type safety |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Graphics |  |  | PropertyList/Dictionary/Map | |
| Base Level | * Pl: PropertyList   # get(s: Slot): Object  # set(s: Slot, o: Object): Object |  | Meta Level | slot1 | value1 |
|  |  |  | slot2 | value2 |
| slot3 | value3 |

**Anything:**

|  |  |
| --- | --- |
| Intent | We need to represent any data that can include sequences of data and should be stored recursively and should be easily extendable. Solution is to implement representation for simple/complex values. |
| Benefits | * Readable and streamable format * Universally applicable, flexible interchange across class/object boundaries |
| Liabilities | * Type safety, unapparent intent * Overhead for value lookup and access / or cast * No real object, everything is just data without interpretation |



# Shepherding Process

**Three Iterations:** How to budget your time and effort to make shepherding effective

**The Shepherd Knows the Sheep:** How to establish a productive relationship between you and the author

**Half a loaf:** How to make sure that shepherding continues to move forward

**Big Picture:** How to grasp the gist of the pattern right off the bat.

**Author as Owner:** How to keep from writing the pattern for the author

**Forces Define Problem:** How to understand the problem at a deeper level

**Why:** Used to improve patterns, Primary focus of PLoP conference, Each writers' workshop contains 5 to 8 papers, Authors must read before the conference.

Giveeach other feedback on their work in a peer review session. The authors of the paper under discussion remain silent.

**Practices:**

1. **Pattern Scanning:** Reading the pattern. Does problem-solution, known uses, context, forces and consequences make sense?
2. **Styling the forces:** Are the forces listed as items? Are forces accurately/coherently described? Why are the given forces important?
3. **“BUT”-Style:** Is “BUT”-style used and does it build tension. Does it lead to “bold face” solution
4. **Detailed Example:** Is there a detailed example and technical diagram? Are there any examples describing the pattern and are those appropriate?
5. **Known Uses:** Are there at least three known and appropriate uses? Are known uses matching the real context/solution of the pattern
6. **Relationship:** Are the related patterns described in a logical order? Are the relationships appropriate?
7. **Standalone:** Does the pattern overlap with other patterns? Is the pattern description coherent? Are there suggestions for improvement?

# QUESTIONS

**Atomic Parameter Pattern:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Intent | Benefits | Liabilities |
| Atomic Parameter Pattern | Two applications with differing technologies need to communicate over a consistent and common interface -> Use of API | send 1 thing | spammy, not expressive without documentation, additional work to map generic types to language types |
| Atomic Parameter List | see above but with more than 1 parameter | more stuff can be sent at once | serialization and deserialization needed, More work, less expressive than single atomic parameters |

* Are Atomic Parameter Lists suitable for all programming languages out of the box? [no]
* Single Atomic Parameter is a suitable pattern for a notification daemon with optional image and response functionality? [no]
* Should the Atomic Parameter Pattern be used to define multiple data units in API request and response messages? [No]
* Should the size of an atomic parameter list be as big as possible? [No]
* Are optional atomic parameters marked as such in the atomic parameter list pattern? [Yes]
* The Atomic Parameter pattern can be applied to the in parameter in a request message. True
* The Atomic Parameter pattern can be applied to the out parameter in a response message. True
* An Atomic Parameter cannot be transported as part of the URI. False
* Wird das Atomic Parameter Pattern bei einer REST-API eingesetzt, kann ein einzelner Parameter im URL-Pfad mitgegeben werden [Ja]
* In Verbindung mit einer REST API werden beim Atomic Parameter List Pattern mehrere Parameter im URL-Pfad mitgegeben [Ja]

**Parameter Tree Patterns:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Intent | Benefits | Liabilities |
| Parameter Tree Pattern | see above but as a tree | hierarchy allowed | needs to be perfectly documented to work, can become too complex |
| Parameter Forest Pattern | trees at same level |  | spikes -> irregular sized tree |
| Pagination Pattern | How to request data that doesn't fit into a specific page -> send partial page | only send some stuff and not everything | consistent application needed, page size requires careful consideration, limited use with little data, big datasizes lead to performance issues |

* Can Parameter Tree Pattern handle multiple trees on the top level for multiple trees? [No]
* Is Pagination Pattern mainly for handling large data that doesn’t fit in a single response? [Yes]
* A parameter forest can consist of multiple parameter trees? [Yes]
* A parameter tree can have more than one root node? [No]
* Pagination helps to reduce the number of requests by sending as much data as possible at once. False
* With pagination, large response data is divided into chunks. True
* With pagination, servers can inform clients about the total number of pages. True
* Können Daten strukturiert nach dem "Atomic Parameter Forest Pattern" mehrere "Parameter Trees" beinhalten? [Ja]
* Soll der Time- oder Token-based Ansatz des Pagination Patterns verwendet werden, wenn sich die Daten ändern können, während der Benutzer sie "durchblättert"? [Ja]

**Version Identifier:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Intent | Benefits | Liabilities |
| Version Identifier | How to define versioning for APIs. Put a fucking number on it | reduces likelihood of issues due to API, version tracing | governance |
| Semantic Versioning | Introduce 2 more digits. 1.2.3 -> 1 is major version (incomp), 2 is extension (comp), 3 is bugfix (comp) | clarity about compatibility | increased governance |
| Two(X) In Production | Two or X API versions supported at once | impact on clients reduced, clear timeframe for clients, ability to rollback | Huge overhead, no guarantee for clients to ever change to new API |
| Limited Lifetime Guarantee | Guarantee API functionality until X | Client has guarantee about functionality | no guarantee for clients to ever change to new API, compatibility breaking bugfixes hard, new features also problematic |

* Should fixes only increment the third part of the semantic versioning? (1.1.0 -> 1.1.1) [yes]
* Semantic versioning makes it easier to differentiate between bug fixes and breaking changes in an API [yes]
* Does the change in semantic versioning from 1.0.0 to 1.1.0 reflect a breaking change? [No]
* Is enforcing deletion policy in the two in production pattern important? [Yes]
* Version Identifiers should only be specified in the API documentation (e.g. OpenAPI). False
* A minor change can be an incompatible change. False
* A patch is always backwards compatible. True
* Does the Semantic Versioning pattern require the full three-number versioning scheme to be exposed as the Version Identifier. [No]

**Game Loop:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Intent | Benefits | Liabilities |
| No sync | game runs as fast as possible |  |  |
| Fixed time step | game runs as fast as defined -> fps lock |  |  |
| Variable time step | game runs with delta\_time. causes indeterminism -> floating point calculations |  |  |
| Fixed time step, variable rendering | lock game logic at defined fps, render with delta\_time |  |  |

* Fixed update time step solves the issue of floating-point errors. [yes]
* Fixed time stamp with no synchronization is the preferred variant by gamers. [no]
* A UI that waits on user input until it progresses further (is blocking) is an application of the game loop pattern [No]
* In the game loop variant "Take a little nap" the game runs at a steady frame rate if the frame-cycles takes longer than the fixed time [No]
* Rendering is part of the game loop. Yes.
* There is only one way to implement a game loop. No.
* Can residual lag be used as a parameter for the rendering. Yes.
* The render() part can be skipped to keep the game time constant.Yes
* Games logic always runs 60 times per second.No

**Game Component:**

|  |  |
| --- | --- |
| Intent | How to achieve modularity with behaviors of a class -> components |
| Benefits | components can be replaced easily (with vtable even at runtime) |
| Liabilities | complexity -> sometimes unknown how things correlate, indirection |

* Is inheritance preferable to composition? [No]
* The component pattern creates more objects which decreases the performance, can data locality help to decrease the performance losses? [Yes]
* In terms of maintainability in the future, is it a good idea to create a single monolithic game class? [No]
* If we implement a messaging system, where each component sends messages to the container which in return broadcasts it to all components, is this an implementation of the Mediator Pattern? [Yes]
* One benefit when using components is that the code base will contain fewer files/classes. False
* A player class in a game is a typical example of a component. False
* Components always communicate with each other through the state of entities. False
* Does the component pattern decouple different domains by moving the domain logic into subclasses? No
* Should the component pattern always be used in performance-critical code? No

**Event Queue:**

|  |  |
| --- | --- |
| Intent | How can classes communicate with each other without a hierarchy chain? -> events |
| Benefits | flexible communication, avoid hierarchy chain, global or custom scope |
| Liabilities | order not guaranteed, reading by recipient not guaranteed, overhead |

* Does an event queue help synchronize actions in a game? [no]
* Does the use of ring buffers allow for continuous adding and deleting of events without the need for shifting elements? [yes]
* In a ring buffer, is the head of the queue where requests are removed from? [Yes]
* Does the pattern require immediate processing of requests upon their receipt? [No]
* The Event Queue is addressing the same problems as the Observer pattern. False
* Event Queues can lead to problems like feedback loops and complexity in handling object lifetimes. True
* Event Queues are mainly useful to facilitate communication between applications. False
* Do event queues make it easier to take advantage of multi-threading? Yes
* Do event queues allow you to make non-blocking calls? Yes

**Whole-Part Pattern:**

|  |  |
| --- | --- |
| Intent | Divide a complex software into smaller parts while the end-user doesn't notice this. |
| Benefits | modularity, re-usability, scalability |
| Liabilities | complexity |

* Fördert das Whole-Part Pattern die Wiederverwendbarkeit von Komponenten? [Ja]
* Wenn man das Whole-Part Pattern zu extrem anwendet und (zu) viele Komponenten hat, kann dies zu Überkomplexität führen? [Ja]
* Beim Whole-Part Pattern können das Whole und das Part auch unabhängig voneinander eine sinnvolle Logik haben. [Nein]
* Beim Whole-Part Pattern können die einzelnen Parts auch direkt verwendet werden. [Nein]
* Ist ein direkter Zugriff auf die Komponenten, aus dem ein Objekt besteht, beim Whole Part pattern möglich? [Nein]
* Hilft das Whole-Part pattern dabei, die Wartbarkeit und Erweiterbarkeit der Software zu verbessern? [Ja]

**Forwarder/Receiver:**

|  |  |
| --- | --- |
| Intent | Abstract IPC communication without broker. Create 2 endpoints, forwarder -> sender and receiver |
| Benefits | IPC performance, modularity with IPC, abstraction, good scalability with load on forwarder/receiver |
| Liabilities | IPC itself on each endpoint is not |

* Wenn das Forwarder / Receiver Pattern eingesetzt wird, kann mit wenig Aufwand die IPC implementiert werden, ist dann aber kaum noch anpassbar. [Nein]
* Der Forwarder bestimmt die Physische addresse der Empfänger über das name-to-address mapping. [Ja]
* Kann ein Peer im Forwarder-Receiver Pattern sowohl als Client als auch als Server agieren? [Ja]
* Ist es im Sinne des Patterns, Speicher-Adressen (z.B. Pointers) hin und her zu senden? [Nein]
* Ist das Pattern mit Variation in einer Microservice-Architektur brauchbar? [Ja]
* In a P2P environment where the peer distribution changes frequently, is it practical to implement the Forwarder-Receiver design pattern? [No]
* A name-to-address must refer to a single address? [No]
* the name of the recipient is contained in the message? [No]

**Blackboard:**

|  |  |
| --- | --- |
| Intent | arcane/nondeterministic problem -> apply knowledge sources one by one |
| Forces | experimentation, modularity of knowledge sources, fault tolerance |
| Liabilities | not reliable, only for testing and experimentation, no parallelization |

* Kommunizieren im Blackboard-Pattern die spezialisierten Module miteinander, um eine Lösung zu finden? [Nein]
* Kann das Blackboard-Pattern verwendet werden, um komplexe und sich schnell ändernde Probleme in der Softwareentwicklung anzugehen? [Ja]
* Control ist verantwortlich für die Entscheidung, welche Knowledge Source auszuführen ist [Ja]
* Ist die Verwendung des Patterns sinnvoll, wenn ein deterministischer Algorithmus bekannt ist? [Nein]
* Kann man sicherstellen, dass eine Umsetzung des Patterns immer eine Lösung für seine Probleme finden wird? [Nein]
* Can a knowledge source solve a problem on its own? [No]
* Can the controler access the blackboard? [Yes]
* Ist das Blackboard-Pattern sinnvoll für Probleme, die keine bekannte Lösungsstrategie haben? [Ja]
* Systeme welche das Blackboard-Pattern nutzen stoppen nur sobald eine acceptable lösung gefunden wurde? [Nein]

**Lazy Acquisition:**

|  |  |
| --- | --- |
| Intent | How to load data when needed and not immediately |
| Benefits | resource efficiency, faster startup, lower overhead, dynamic resource availability |
| Liabilities | extra resource loading required |

* Das Lazy Acquisition Software Pattern zielt darauf ab, alle Ressourcen zu Beginn der Programmausführung zu erwerben. [Nein]
* Das Lazy Acquisition-Muster kann zur Verbesserung der Startleistung beitragen, indem die Beschaffung von Ressourcen verschoben wird, bis sie tatsächlich benötigt werden. [Ja]
* Ist der || Operator in Java ein Beispiel von Lazy-Acquisition? [Ja]
* Ist Lazy-Loading sinnvoll, wenn alle Daten von Anfang an gebraucht werden? [Nein]
* Ändert sich die Verwendung einer Ressource, wenn ich Lazy-Acquisition einführe? [Nein]
* Lazy Acquisition reduziert die Aufstartzeit eines Programms, weil die Ressourcen erst kurz nach der Initialisierung geladen werden. [Nein]
* Der Zugriff auf eine Ressource mit Lazy Acquisition ist stabil in der Ausführungszeit. [Nein]
* The resource proxy will acquire the resource only when the resource is actually accessed by the end user [NO]
* The resource proxy is not only responsible for resource acquisition, it can also do resource management. eg.caching, additioanl control features. [YES]

**Coordinator:**

|  |  |
| --- | --- |
| Intent | coordinate clients to first dry-run their task and then execute, if one fails, all stop their execution |
| Benefits | consistency, atomicity, location transparency, scalability, transparency |
| Liabilities | overhead -> dry-run then execute, overhead by registering on coordinator |

* Die Arbeit jeder Teilnehmer wird in prepare und commit unterteilt. [Ja]
* Der Einsatz des Patterns führt zu overhead, weil der Task in vier Phasen unterteilt wird. [Nein]
* Does the client directly ask all participants about task completion capability? [No]
* Must all participants succeed for the system to change? [Yes]
* Is the client asking all participants if they are able to complete the task? [No]
* Do all participants need to succeed for the system to change? [Yes]
* If something goes wrong during the commit phase (without using the three-phase commit variant). Are changes rolled back so that the system remains in a consistent state? [NO]
* Since each participant has indicated in the prepare phase that the work will succeed, the commit phase will for sure be successful? [No]

**Resource Lifecycle Manager (RLM):**

|  |  |
| --- | --- |
| Intent | how to separate resource management away from clients -> introduce resource manager which handles this interaction for clients |
| Benefits | Availability, Scalability, Complexity, Performance, Stability, transparency |
| Liabilities | single point of failure, flexibility |

* Erfordert der Ressource Lifecycle Manager, dass der Ressourcennutzer sich mit den Komplexitäten der Ressourcenverwaltung auseinandersetzt? [Nein]
* Hilft der Ressource Lifecycle Manager die Effizienz der Ressourcennutzung in einer Anwendung zu verbessern? [Ja]
* Gibt es eine Variante des Resource Lifecycle Managers, bei dem die Strategie zum Managen der Resource konfigurierbar ist? [Ja]
* Sind Inversion of Control Containers (z.B. BeanFactory (Java), Autofac (C#), etc.) beispiele eines Resource Lifecycle Managers? [Ja]
* Der Resource Lifecycle Manager garantiert die optimale Verwendung seiner Resourcen [Nein]
* Kann ein Resource User direkt auf eine Resource zugreifen? [Ja]
* Es gibt immer genau einen RLM in diesem Pattern? [Nein]
* This pattern can increase the load that an application can handle by using caching? [YES]
* Does the Resource Lifecycle Manager handle the removal or decommissioning of resources when they are no longer needed? [YES]