Introduction to Patterns

Definition

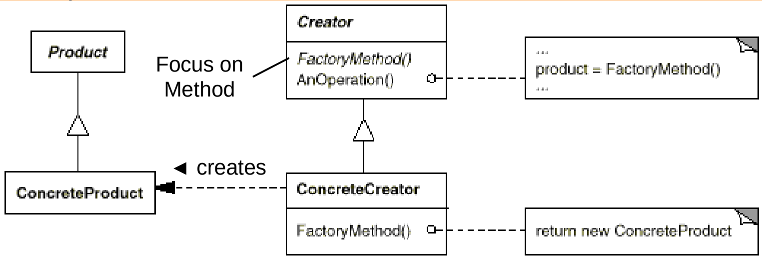
Descriptions of successful engineering stories with well-known *intent*, addressing recurring *problems* with a generic *solution*. Patterns include *forces*, *benefits*, *liabilities*, *names* and *implementations*.

Pattern Relations (No Pattern is an Island)

Refinement: Patterns relate on others for their implementation
Setting Context: Patterns are pre-requisites for others
Variation: Patterns show alternative or variant solutions
Specialization: Patterns are special variants of more general patterns
Pattern Languages: Combine patterns according to these relationships

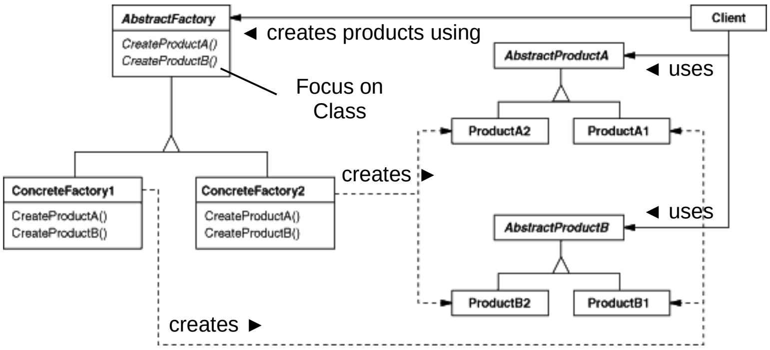
Creational Patterns

Factory Method



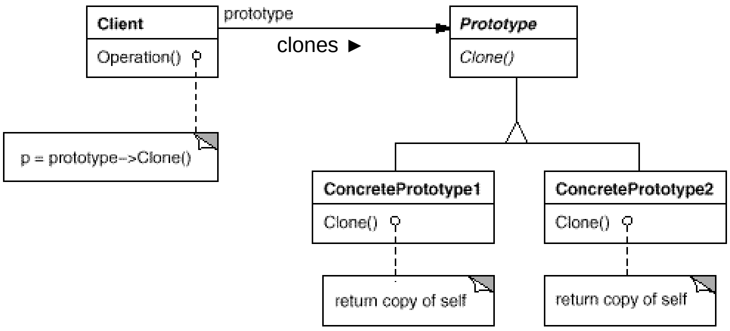
Intent: Define an interface for creating an object (Product), but let subclasses (ConcreteCreator) decide which class to instantiate (ConcreteProduct). Lets a class defer instantiation to subclasses.

Abstract Factory



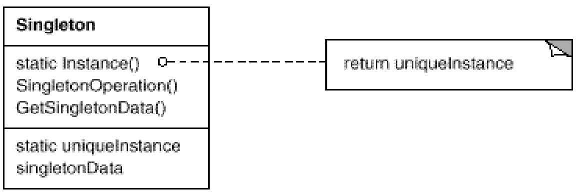
Intent: Provide an interface (AbstractFactory) for creating families (ConcreteFactory) of related or dependent objects (AbstractProduct) without specifying their concrete classes (ConcreteProduct).

Prototype



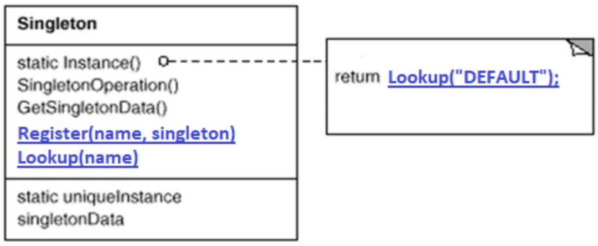
Intent: Specify the kinds of objects to create using a prototypical instance (e.g. red circle, blue square), and create new objects by copying this prototype using it's abstract interface.

Singleton (Do never use)



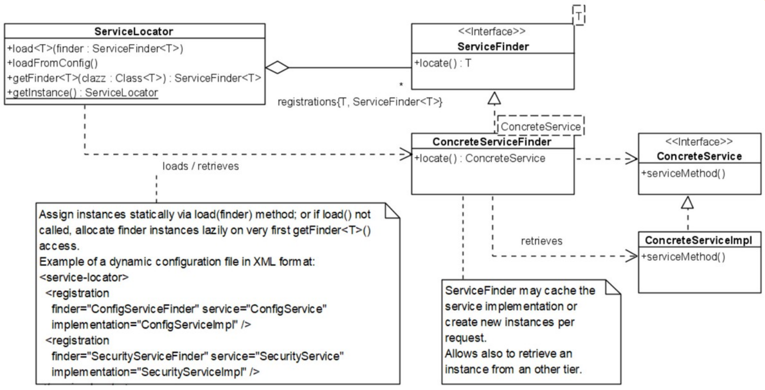
Intent: Ensure that a class has only one, globally accessible instance
Problem: OO-languages usually don't restrict number of instances
Solution: Use a private constructor and add a single static factory method for instance access. Allows lazy initialization and sub-classing
Benefits: Controlled access to sole instance, "more flexible"
Liabilities: Global variable/state, tight coupling, problematic with multi-threading, prevents polymorphism, limits unit-testing and expandability
Relations: Setting Context by *Class Factory Method, Lazy/Eager Acq.*

Registry (Singleton Variation)



Solution: Use global registry to register *Singleton* instances.
Benefits: Same as *Singleton*, but more flexible and allows mocking
Liabilities: Same as *Singleton*, but doesn't control instantiation

Service Locator (Singleton Variation)



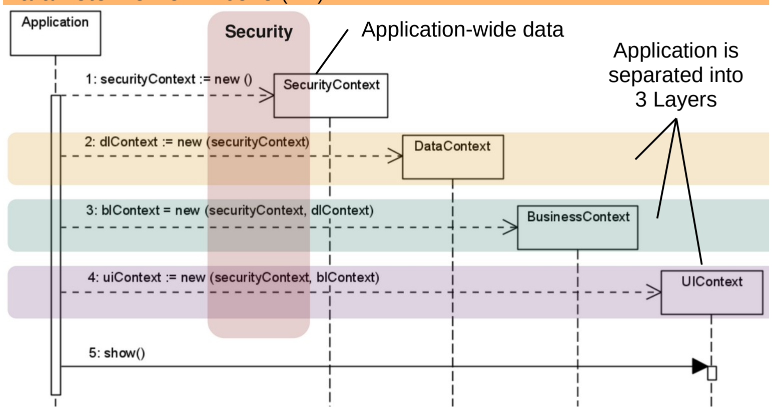
Intent: Provide global instances while maintaining a loose coupling
Problem: Global instances (*Singeltons*) are not easily exchangeable
Solution: Create a global ServiceLocator that holds ServiceFinders, which can then be used to locate ConcreteServices (instances)
Benefits: Only one *Singleton*, interfaces allow abstraction/testability
Liabilities: ServiceLocator is still a *Singleton*, problems not solved
Relations: Setting Context by *Abstract Factory*, Variation of *Registry*

Singleton Assassination

Monostate (Borg)

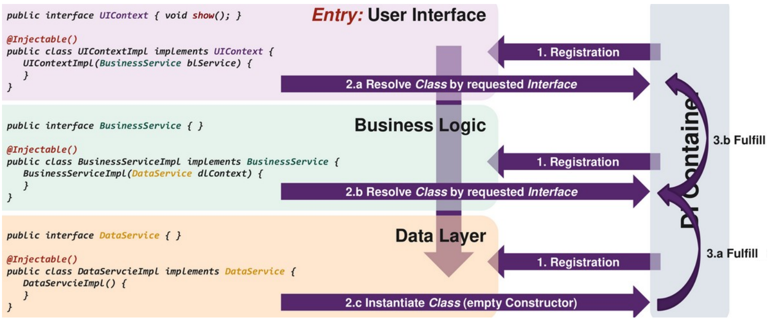
Intent: Provide multiple instances that behave like a single instance
Problem: Different instances of the same class have their own state
Solution: Create a *Monostate* class in which all member variables are static, use non-static methods to access these variables
Benefits: Transparency, polymorphism, allows wrapping *Singletons*
Liabilities: Shared state can cause unexpected behavior, breaks inheritance hierarchy (non-monostate to monostate not possible)
Example: `public int getX(){ return Singleton.instance().getX() }`
Relations: Alternative to *Singleton*

Parameterize from Above (PfA)



Intent: Provide application-wide data without making it global
Problem: Some data needs to be available in all layers/components
Solution: Parameterize each layer/component from above
Implementation: Inject "global" data during the bootstrapping process
Benefits: No global variables, exchangeable implementations, SoC
Liabilities: Increases complexity, context must be passed through entire application, Fragile bootstrapping (entire wiring on start-up)

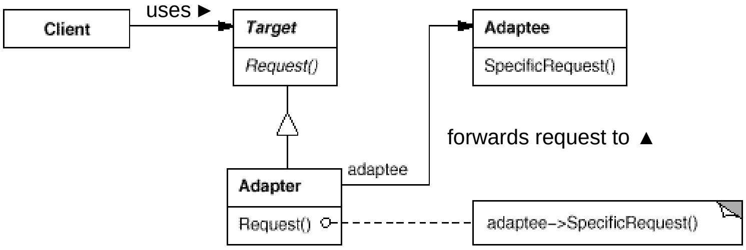
Dependency Injection



Intent: Provide automatic wiring of data/components used in application
Problem: Manual wiring is fragile/inflexible and creates code bloat
Solution: Load, instantiate and wire components dynamically at start-up
Implementation: Create a central DI-Container, declare required dependencies in class using interfaces, register actual implementations in DI-Container (e.g. annotations), DI-Container handles instantiation
Benefits: Low coupling, based on interfaces, flexible, allows “Singleton”
Liabilities: Adds black magic, recursive dependencies can prevent start-up, no compiler support (Reflection), debugging can be harder
Relations: Setting Context by *PfA*, *Service Locator*, *Registry*

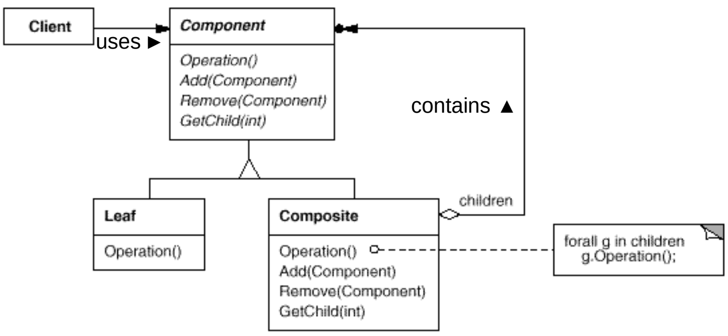
Structural Patterns

Adapter



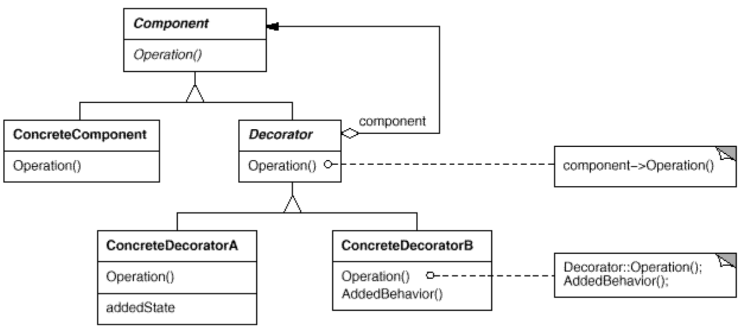
Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces (Interoperability).

Composite



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

Decorator

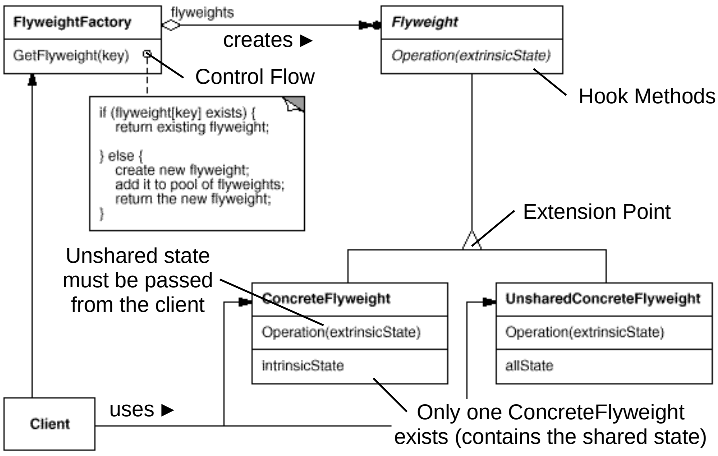


Intent: Attach additional responsibilities (Decorators) to an object (Component) dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Pooling

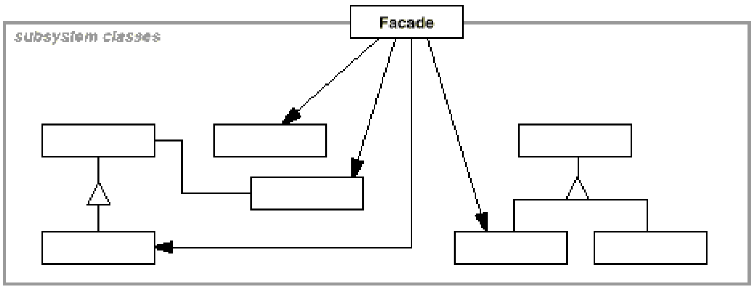
Intent: Allow recycling of resources to prevent costly acquisition/release
Problem: Acquisition/release of resources happens often and is slow
Solution: Manage multiple resource instances in a pool
Implementation: Define the max. number of resources in the pool, load them lazy/eager and determine their recycling/eviction mechanics.
Benefits: Simple, predicable and efficient resource acquisition/release
Liabilities: Adds management overhead, requires synchronization
Relations: Setting Context for *Flyweight*

Flyweight



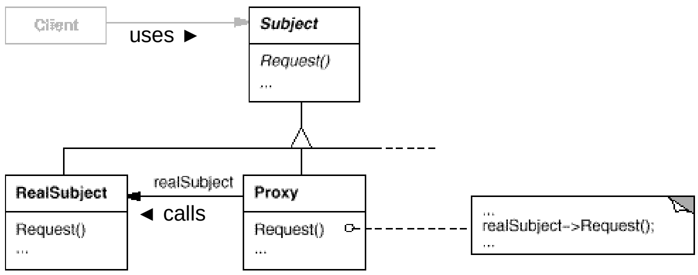
Intent: Avoid multiple copies of identical (constant) objects
Problem: Having identical objects increases storage cost unnecessary
Solution: Use sharing to support large numbers of objects efficiently
Implementation: Separate objects into shared (intrinsic) and unshared (extrinsic) data, use a manager (FlyweightFactory) to create shared, immutable objects (Flyweights) while preventing multiple instances
Benefits: Reduces the total number of instances (saves storage space)
Liabilities: Flyweights are *Value Objects* (no identity), more run-time costs
Relations: Setting Context by *Immutable Value, Singleton, Class Factory, Pooling,* Combination with *Composite* is common
Critique: Combines too many Patterns into one, has no “real” solution

Facade



Intent: Provide a unified interface (Facade) to a (sub) set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use (Simplicity).
Problem: Complicated interfaces, tight coupling to subsystem
Solution: Provide a unified interface to interfaces in the subsystem
Implementation: Create one or more Facades that wrap the subsystem
Benefits: Isolation of code, abstraction of (volatile) subsystem
Liabilities: Facade can become a god object coupled to all classes
Relations: Variations are *Adapter*, *Proxy*

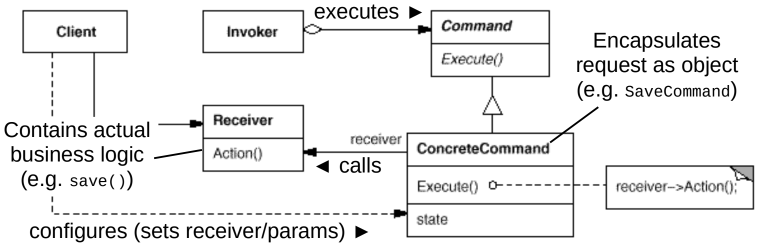
Proxy



Intent: Provide a surrogate or placeholder (Proxy) for another object (RealSubject) to control access to it.

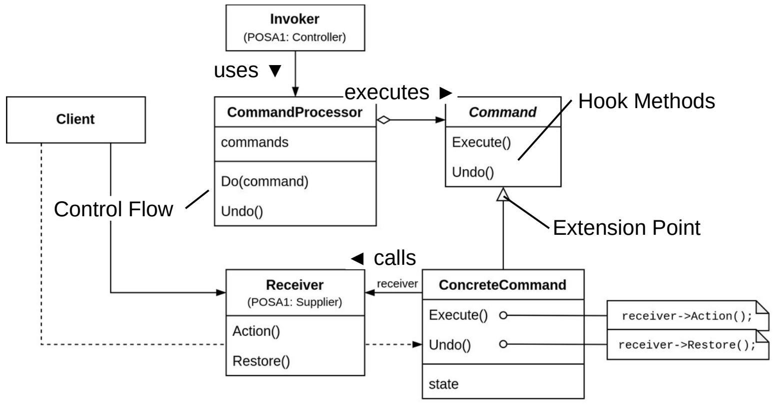
Behavioral Patterns

Command



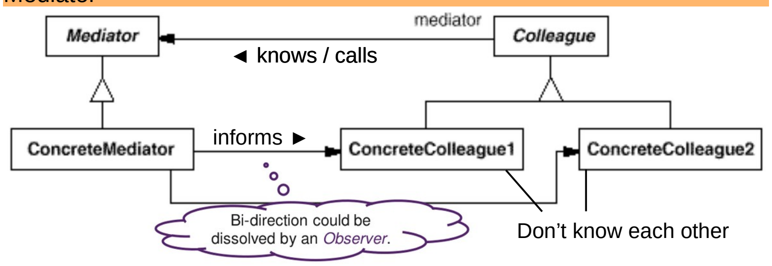
Intent: Encapsulates requests (i.e. methods) as objects so they can be parameterized, scheduled, logged and/or undone
Problem: Direct method execution creates tight/immediate coupling
Solution: Encapsulate requests/methods in Commands
Benefits: Commands are reusable/extendable, allows decoupling
Liabilities: Can introduce a large number of Command classes
Relations: Setting Context for *Command Processor*, *Internal Iterator*, Setting Context by *Strategy*

Command Processor (POSA1)



Intent: Manage Commands so their execution can be deferred/undone
Problem: Request and execution of command are tightly coupled
Solution: Create a Command Processor that manages Commands, allowing them to be scheduled and undone in one central location
Implementation: Command Processor has command stack with history
Benefits: Flexibility, Processor and Controller are separated, allows additional services for Command execution, enhances testability
Liabilities: Efficiency loss due to additional indirection
Relations: Setting Context by *Command*, *Memento*

Mediator

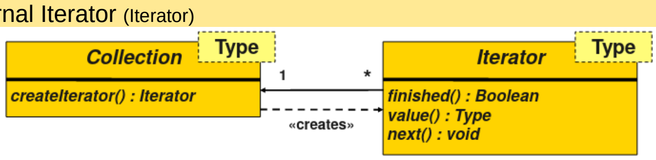


Intent: Promotes loose coupling by preventing explicit object referral
Problem: Strong coupling of objects and complex communication
Solution: Introduce *Mediator* that encapsulates object interactions
Implementation: Mediator as Observable, Colleagues as Observers
Benefits: Low coupling, re-usability, centralized communication
Liabilities: Complexity, Single Point of Failure, (Without *Observer*: Hard maintainable monoliths, limits sub-classing of Mediator)
Known Uses: Message Bus Systems, Redux Dispatcher
Relations: Refinement by *Observer*

Iterator Patterns

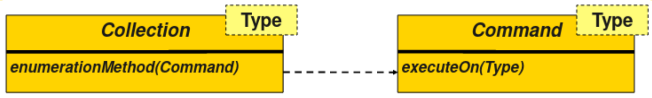
Intent: Avoid strong coupling between collection and iteration
Problem: Iteration-mechanism depends on collection implementation

External Iterator (Iterator)



Solution: Iteration-mechanism is encapsulated in a separate object
Implementation: 4 methods: `create`, `finished`, `value`, `next`. In Java represented with `list.iterator()`, `hasNext()`, `next()` (includes `value`)
Benefits: Single interface to iterate through any kind of collection
Liabilities: Shared access difficult, requires life-cycle management of iterator, strong coupling between iterator and collection
Relations: Setting Context by *Plain Method Factory*

Internal Iterator (Enumeration Method)

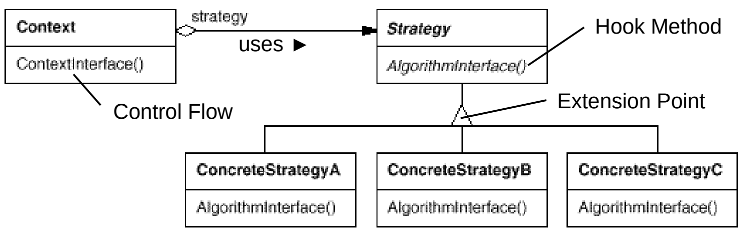


Solution: Place the responsibility for iteration on the collection
Implementation: Use *Command* that is applied to each element in collection. Modern languages use lambdas (Java: `list.forEach(...)`)
Benefits: No loop-logic in client, synchronization at traversal-level
Liabilities: Some people think functional approaches are "too complex"
Relations: Setting Context by *Command*

Batch Method

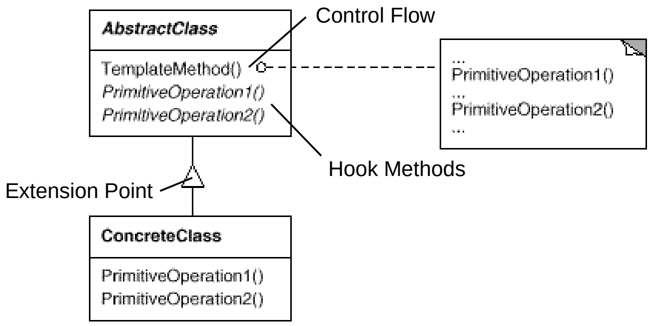
Problem: Collection and iterator user are not on the same machine
Solution: Group multiple collection accesses together
Implementation: Client-side data structure groups multiple collection accesses together and calls remote collection when appropriate
Benefits: Reduces the communication overhead
Liabilities: Increases complexity on client side
Examples: String Builder, SQL Cursors, Pagination
Relations: Variation of *Remote Proxy*

Strategy



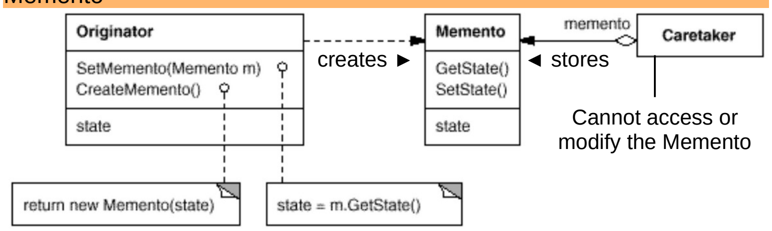
Intent: Define a family of algorithms (Strategy), encapsulate each one (ConcreteStrategy), and make them interchangeable. Strategy lets the algorithm vary independently from clients (Context) that use it.

Template Method



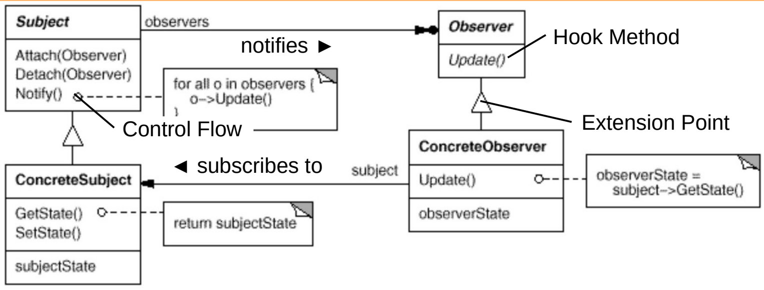
Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. *Template Method* lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Memento



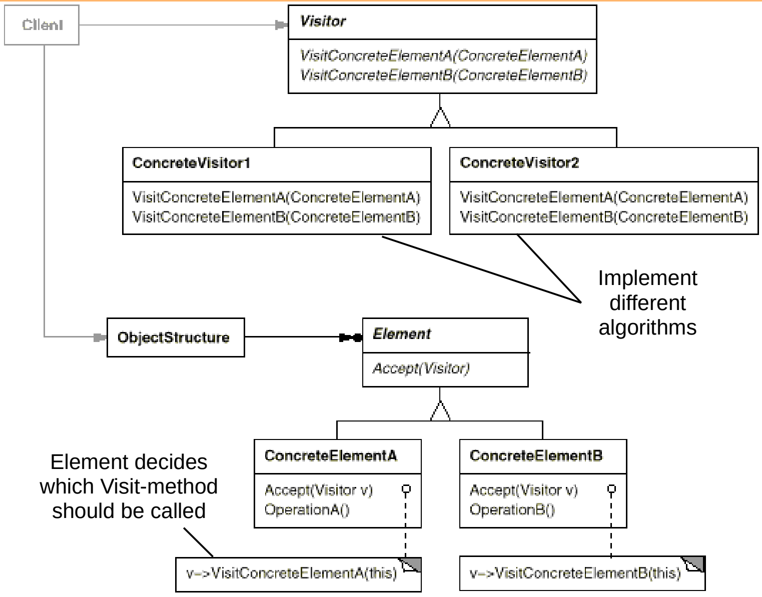
Intent: Capture an object's internal state without violating encapsulation
Problem: Objects encapsulate their state, making it inaccessible
Solution: Capture an object's internal state in a *Memento*
Implementation: Originator creates Memento and uses it to (re)store it's internal state. Caretaker stores (i.e. serializes) the Memento.
Benefits: Allows saving/restoring state without violating encapsulation
Liabilities: Memory inefficient (copy of state), no direct access to state, requires language support (C++: `friend`, Java: `package-private`)
Known Uses: Serialization, Save States (Reflection is more modern)
Relations: Setting Context by *Factory Method*, for *Command Processor*

Observer



Intent: Define a one-to-many dependency between objects so that when one object changes state (Subject, Observable), all its dependents (Observers) are notified and updated automatically.

Visitor

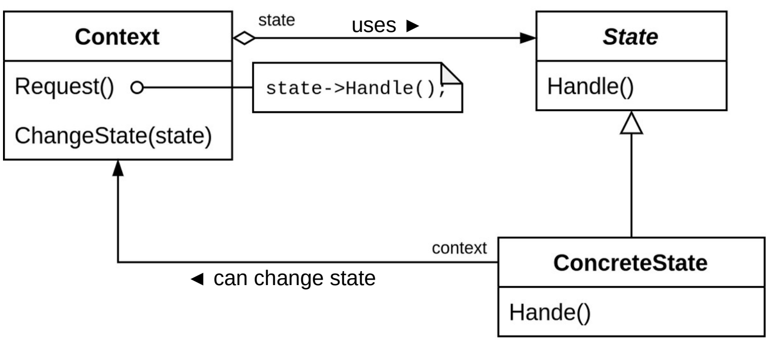


Intent: Separates algorithms from the objects they operate on
Problem: Algorithms must be added to classes without modifying them
Solution: Add the algorithm to a separate class instead of the original
Implementation: Visitor declares Visit-method for each concrete element. Element accepts Visitor and calls matching Visit-method. This is called Double-Dispatch (`Visitor` → `Element` → `Visitor`)
Benefits: Separates unrelated logic, allows adding more algorithms
Liabilities: Adding new Elements requires modifying Visitor, cannot access private fields, visiting sequence is defined within Elements
Relations: Combines with *Composite*, *Strategy*, *Iterator*, *Chain of Responsibility*, *Interpreter*

State Patterns

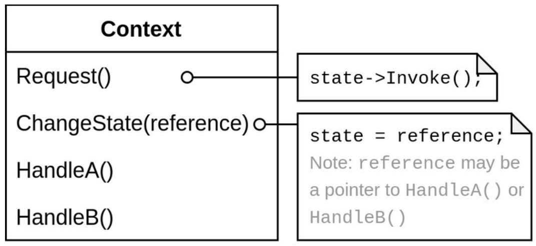
Intent: Change behavior of an entity based on it's state at run-time
Problem: Behavior changes can lead to large conditional statements
Note: State Patterns focus on the representation of states and less on their transitions. Transitions below are included to improve clarity.

Objects for State (State)



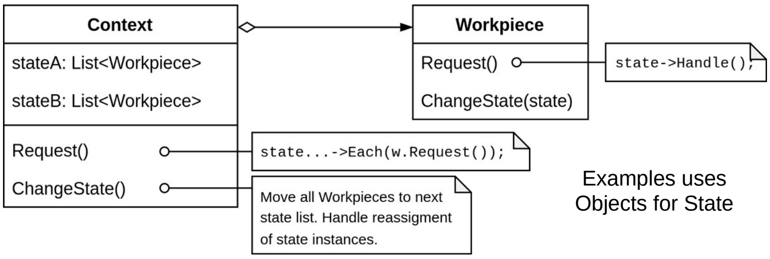
Solution: Create one class per state and use context for state transition
Implementation: Allow ConcreteState to change state in Context
Benefits: Removes conditional statements, allows extension of states
Liabilities: Complex, overkill for few states, distributes behavior
Relations: Specialization of *Strategy*, Setting Context for *Interpreter*

Methods for State



Solution: Represent states as one (or more) method references
Implementation: Add all methods on Context, transition using pointers
Benefits: Moves all behavior in Context, reduces number of classes
Liabilities: Context can become unmanageable, two-level indirection
Relations: Setting Context for *Interpreter*

Collections for State

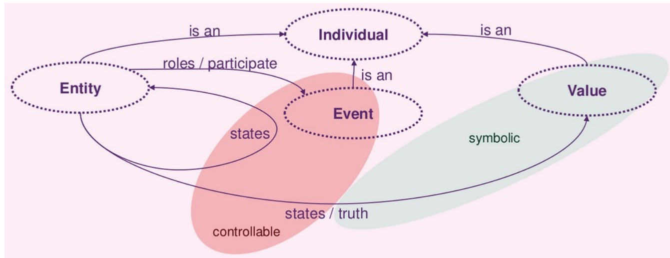


Intent: Manage multiple instances of objects in the same state
Problem: Objects/Methods for States handle state on single instance
Solution: Separate state management from instances using collections
Implementation: Create collections that represent Workpieces in one state. Delegate requests to Workpieces in collections. Create transitions by moving a Workpiece to the next list. Implement state logic using *Objects/Methods for States* or simple conditionals.
Benefits: Optimized for multiple objects, allows various state logic
Liabilities: State manager (Context) can become complex

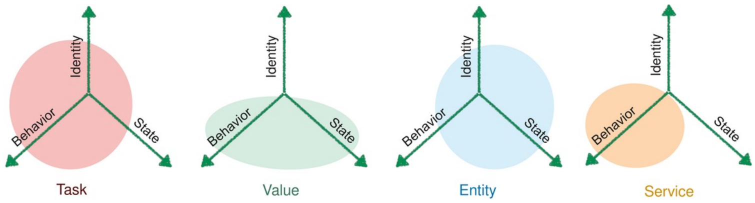
Value Patterns

System Analysis (OOA) (Problem Analysis and Structure)

Individual: Something that can be named and reliably distinguished
Events: Are *individual happenings* at some particular point in time
Entities: Are *individuals* that persist and change over time
Values: Are *intangible individuals* that exist outside time and space



Software Design (OOD) (Patterns of Value)



Entity: Persistent, distinguishable system information with identity
Services: System activities distinguishable only by behavior
Value: Transient content with no identity, behavior in terms of state
Task: System activities with behavior, identity and state (e.g. thread)

Values in Programming (Value Objects)

Characteristics: Values are types with range and dimension but no identity. They are fine-grained, type-safe and repetitive information
Problem: OO languages need to mimic types with object classes (like Date, Point, ISBN, etc.). This creates identity (i.e. references/pointers)
Solution: Use Value Patterns (or language features like `struct`)

Whole Value

Intent: Express the type of a quantity as a Value Class (e.g. Year)
Problem: Primitive quantities (e.g. int) have no real domain meaning
Solution: Wrap primitives in class to add meaning and type-checking
Implementation: Extract primitives with meaning into a `final` class
Example: `final class Year { public Year(int i) { /* ... */ } }`

Value Object (Value Class)
Intent: Allow values to be represented as objects without identity
Problem: Objects are distinguishable by their identity (references)
Solution: Override methods in objects related to identity (e.g `equals`)
Implementation: Override `equals`, `hashCode`, implement `Serializable`
Example: `@Override public boolean equals(Object o) { /* ... */ }`

Conversion Method
Intent: Allow Value Objects to be converted into other Value Objects
Problem: Related Value Objects cannot be used together (type-check)
Solution: Provide a constructor (`Year(int i)`), a conversion instance method (`y.toInt()`) or a Class Factory Method (`Year.fromInt(int i)`)

Immutable Value
Intent: Prevent side-effects when sharing/aliasing Value Objects
Problem: Values can only be copied, but Value Objects can be shared
Solution: Set internal state at construction and prevent modification
Implementation: Declare all fields `private final`, mark class as `final`
Example (Java 16): `record Year(@GreaterThanZero int I) {}`
Relations: *Mutable Companion*, *Class Factory Methods*

Enumeration Value
Intent: Represent a fixed set of constant, type-safe values
Problem: Whole Values do not enforce a constant range of objects
Solution: Define Enum. Values as public read-only fields in a class
Example: `class Enum { public final Month january = new Month(1) }`
Or Built-in: `enum Month { JANUARY(1), /* ... */ }`

Copied Value and Cloning
Intent: Make values modifiable without affecting the original object
Problem: Modifying a Value Object changes all it's shared references
Solution: Create a copy whenever the Value Object must be modified
Implementation: Clone the Value Object using a method (`clone()`)
Benefits: Can imitate the call/return-by-value behavior of value types
Liabilities: Can result in immense object creation overhead
Relations : Setting Context for *Prototype*

Copy Constructor
Intent: Copy Value Objects without using a `clone`-method
Problem: Polymorphic methods (like `clone()`) are not always desired
Solution: Create a copy constructor consuming instance of same type
Example: `final class Year{ public Year(Year y) { /* ... */ } }`

Class Factory Method (Static Factory / Simple Factory)
Intent: Simplify and optimize the construction of Value Objects
Problem: Construction of Value Objects can be expensive/complex
Solution: Use static methods instead of (now private) constructors
Benefits: Allows caching and other optimizations at construction
Example: `class Year{ public static Year fromInt(int i) }`
Relations : Variation of *Flyweight,* Setting Context for *Singleton*

Mutable Companion
Intent: Allow mutations of immutable values using a separate class
Problem: Mutation of immutable values requires complex construction
Solution: Implement a companion that contains modifier methods
Implementation: Create a separate class that takes an immutable value, provides modifier methods and is able to return new mutations
Example:
```
public final class YearCompanion {
    private int value = 0;
    public YearCompanion(Year y){ this.value = y.value; }
    public void next(){ this.value++; }
    public Year asYear(){ return new Year(this.value); }
```
Relations: Refinement by *Plain Factory Method*, *Combined Method*

Relative Values
Intent: Allow relative comparisons of Value Objects (e.g. less than)
Problem: Value Objects are compared by their identity (reference)
Solution: Implement comparison methods on Value Object
Implementation: Implement `Comparable<T>`, `compareTo` and `equals`, forward `equals(Object o)` to `equals(T o)` (called "Bridge Method")

CHECKS Patterns
Intent: Ensure good input is separated from bad input

Meaningful Quantities

Exceptional Behavior
Intent: Handle exceptional behavior without throwing errors
Problem: Missing/incorrect data must be handled by domain logic
Solution: Use distinguished values like `null`, Enums or Optionals to represent exceptional circumstances. Produces *meaningful* behavior
Benefits: Allows errors to be handled using proper domain logic
Liabilities: Can produce complex domain logic for error handling

Meaningless Behavior
Intent: Handle exceptional behavior with minimal domain logic
Problem: Error handling produces more complex domain logic
Solution: Write methods with minimalistic concern for possible failure
Benefits: Error handling does not clutter method
Liabilities: Domain meaning of errors is lost (API fault? User fault?)

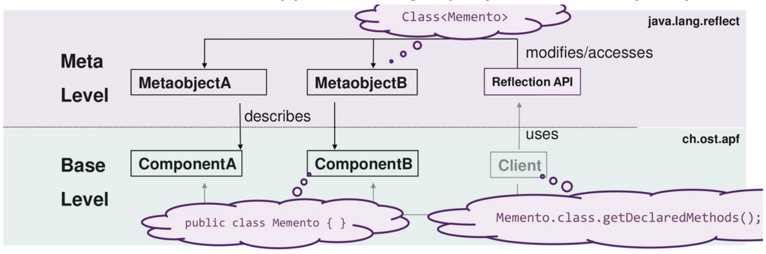
Data Manipulation
Error Back: Inform users of success entering values. Provide read back facility with sanitized values of any information written into domain model. **Visible Implication:** Display computed values immediately along side those that changed. **Deferred Validation:** Defer detailed / structural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

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

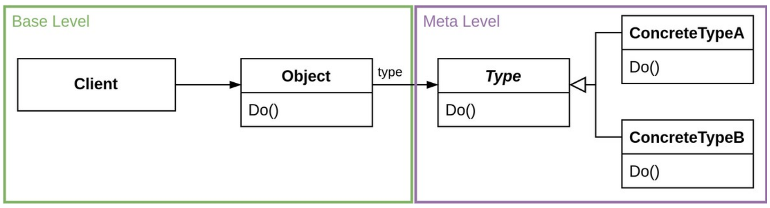
Meta Patterns

Reflection
Definition: A (mainstream) technology for Meta Programming
Problem: Integrating or changing software at runtime is hard
Solution: Allow software to observe and change it's own state (e.g. load DLL, invoke method) by providing a Meta layer
Benefits: Flexibility, Adaptability, Generality (needed by Frameworks)
Liabilities: Non-transparent APIs, limited type-safety/efficiency (late binding), over-engineering, undermines security, complex configs
Usage: Dependency Injection, OR-Mappers, Serialization, Plugins

Components of Reflection (POSA1)
Introspection: Ability of a program to observe/reason about it's state
Intercession: Ability of a program to alter it's state or interpretation
Meta Level: Provides self-representation and modification
Base Level: Defines the application logic (may use meta objects)



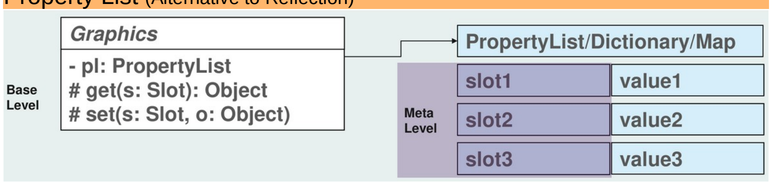
Type Object (Alternative to Reflection)



Intent: Categorize and exchange common behavior at runtime
Problem: Behavior and identification of instances are based on class
Solution: Categorize objects by another object instead of it's class
Implementation: Delegate calls to the (now changeable) type object
Benefits: Extendable types, allows multiple "meta-levels"
Liabilities: Potentially confusing classes, tricky adaption to database
Relations: Setting Context by *Strategy*, Variation of *State*

Item-Descriptor Pattern (OOA)
Intent: Extract common attributes of an object into a separate object
Difference (Type Object): Object represents a description, not a type

Property List (Alternative to Reflection)



Intent: De-/Attach attributes at runtime across class hierarchies

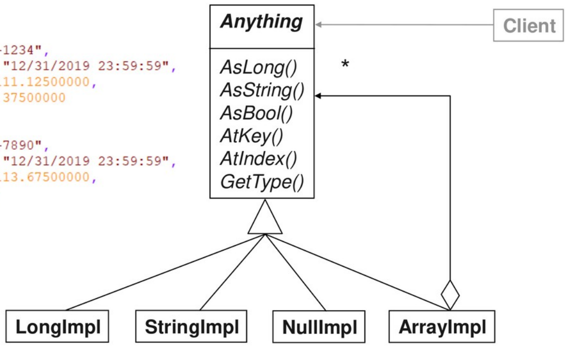
Problem: Changeable, class-independent attributes are not possible
Solution: Provide a property list that maps names to values/objects
Benefits: Allows extending, iterating and sharing attributes at runtime
Liabilities: Inconsistent and unchecked (type, name) attribute access

Bridge Method

Intent: Extend *Property List* with consistent naming and type-safety
Problem: *Property List* has unchecked (type, name) attribute access
Solution: Define a bridge method with fixed name and return type
Example: `String getName() { return (String)props.get("name"); }`

Anything


```
[
  {
    "trackid": "AA-1234",
    "reported_dt": "12/31/2019 23:59:59",
    "longitude": -111.12500000,
    "latitude": 33.37500000
  },
  {
    "trackid": "BB-7890",
    "reported_dt": "12/31/2019 23:59:59",
    "longitude": -113.67500000,
    "latitude": 35
  }
]
```

Intent: Allow mappable data with primitives, sequences and nesting
Problem: Generic and extendable data structures are complicated
Solution: Create a self-describing abstraction for structured values
Implementation: Create one interface that represents all values, create implementations that handle value conversion (if possible)
Benefits: Streamable format that can be applied universally
Liabilities: No "real" object, less type-safe, unapparent intend
Relations: Specialization of *Composite*, Setting Context by *Property List*

Frameworks

Intent: Avoid re-inventing the wheel and reuse what is there
Benefits: Less work, more reliable/consistent code, focus on domain

Framework (Definition)

A set of classes that provide hooks for extensions. In contrast to a library, a framework keeps the control flow ("Inversion of Control")
Examples: Hibernate, Velocity, .NET EF, React.js, Vue.js, MFC

Application Framework (Definition, extends Framework Definition)

Large frameworks used to create families of similar applications. The `main()` method lives in the framework. Are often evolutionary and have very high quality requirements. Use configuration and combination
Examples: Spring, J2EE, ASP.NET, Angular, Office, Apache httpd, Qt

Micro Framework Patterns

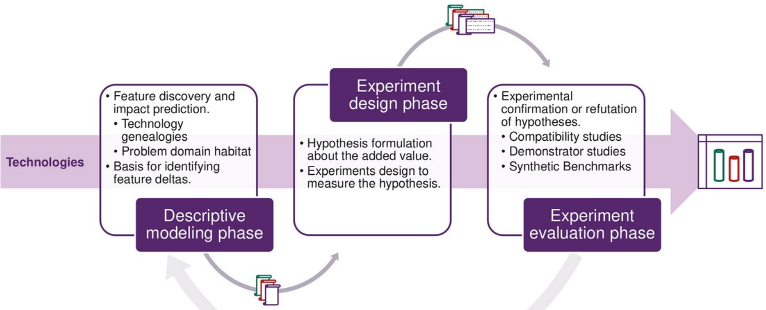
Many patterns have framework characteristics (i.e. Hooks, Extension Points and a Control Flow): *Template Method*, *Strategy*, etc.

Framework Lock-In

Problem: Application implements framework's code, which leads to strong coupling and hinders portability, testability and evolution
Portability: It's hard to migrate to new frameworks or environments
Testability: It's hard to write unit-tests using framework components
Evolution: It's hard to guarantee that the framework will not break
Solution: Evaluate a framework before getting locked-in

Meta Frameworks (Framework Evaluation Frameworks)

What to consider: Acquisition cost (time/money), long-term effect (on product), training and support, future technology plans (of company), response of competitors
Key Ideas: Understand the differences between technologies (Deltas) and how they address the specific usage context



Frameworkers Dilemma

Problem

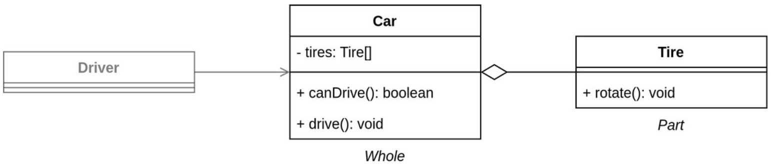
Frameworks need evolutionary improvements, stagnation is often disliked (dead framework!?). However, evolution is challenging:
1. If no one uses the framework, there's no need for improvement
2. If people use the framework, we may introduce breaking changes

Resolution

1. Think very hard up-front: Requires experience and concrete ideas of what to abstract/generalize. **2. Don't care too much about framework users:** Break applications, but provide features/reasons that make porting reasonable. **3. Let framework users participate:** Be fair with your users (e.g. use deprecation instead of deletion).
4. Use helping technology: Make simple, flexible and configurable frameworks, rely on abstractions, encapsulate parameters, try to prevent sub-classing, use good Patterns (*Reflection*, *Property* List).

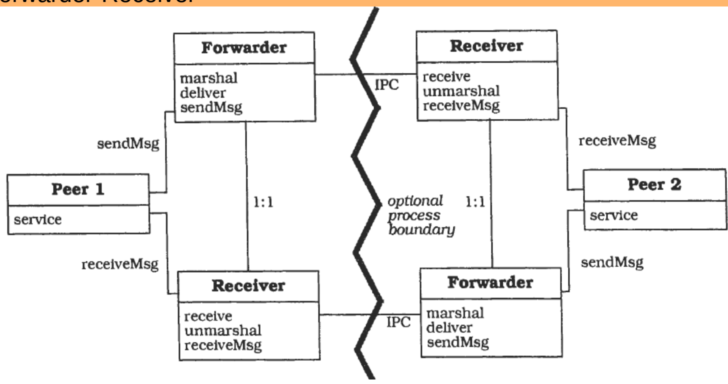
POSA 1/3

Whole-Part



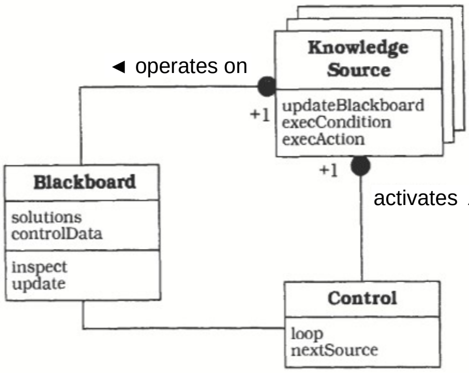
Intent: Combine several objects (Parts) into one semantic unit (Whole)
Problem: Complex objects need to be decoupled without exposing it
Solution: Create a Whole object that contains and coordinates multiple Part objects, hide the Parts from outside access (make them `private`)
Benefits: Separation of concerns, allows reusable/exchangeable Parts
Liabilities: Creating meaningful ("correct") Parts is not always easy

Forwarder-Receiver



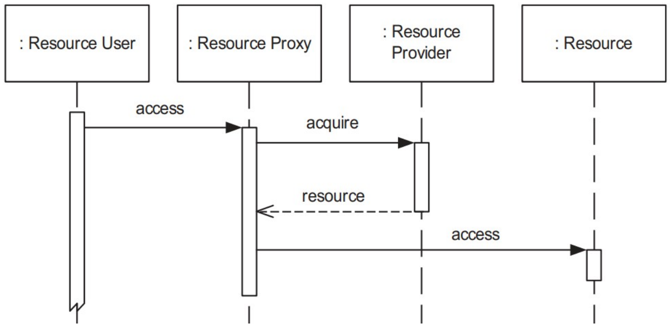
Intent: Decouple peers from their communication mechanism
Problem: Different peers need to communicate using volatile protocols
Solution: Create a Forwarder that can send data to a known Receiver, Peers use the Forwarder/Receiver to communicate with other Peers
Benefits: Encapsulates the communication mechanism/protocol
Liabilities: No flexible reconfiguration, Receiver needs to be known

Blackboard



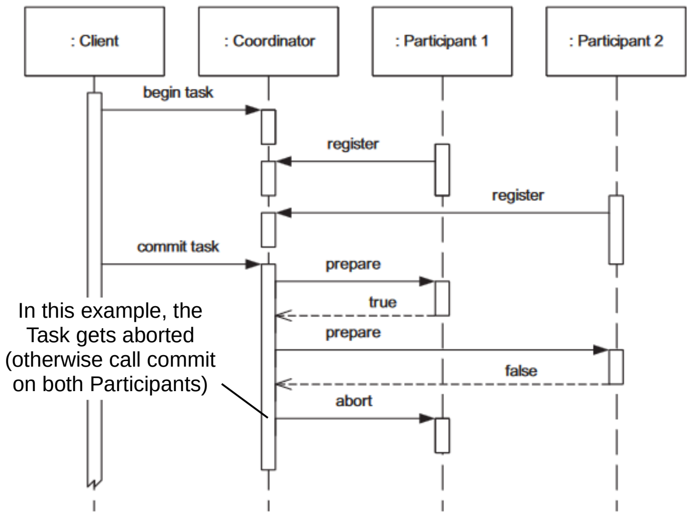
Intent: Allows (partially) solving problem in a non-deterministic domain
Problem: Some problems do not have a clear, deterministic solution
Solution: Create a Blackboard as central data store, allow independent KnowledgeSources to read and modify the Blackboard, each KS knows how to solve one aspect of the problem, use a Control to loop through the KS until a satisfiable solution (determined by Control) is reached
Benefits: Flexible and modular approach to solve complex problems
Liabilities: Very complex, satisfiable solution may never be reached

Lazy Acquisition



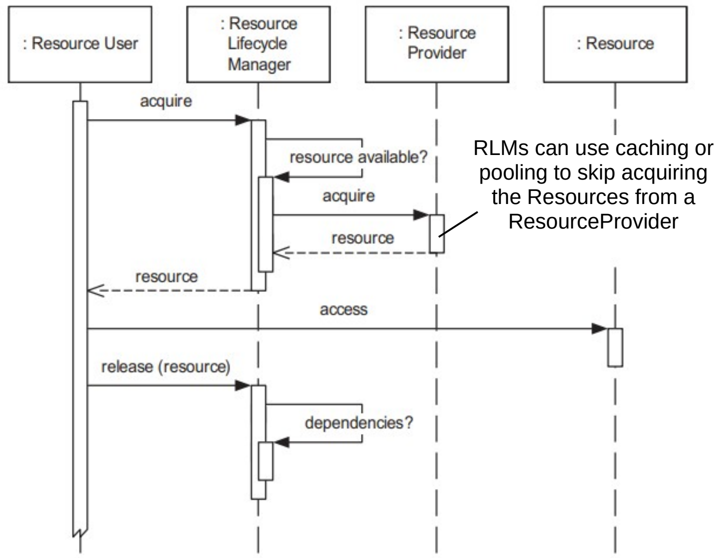
Intent: Defer resource acquisition until required to optimize resource use
Problem: Acquiring resources up front causes (unnecessary) overhead
Solution: Defer resource acquisition to the latest possible time. Instead of a Resource, a ResourceProvider returns a ResourceProxy that only acquires the actual Resource when it's accessed
Benefits: Avoids needless resource acquisition, optimizes start-up time
Liabilities: Unpredictable acquisition time, overhead on first access

Coordinator



Intent: Ensure actions are only performed if all participants can do so
Problem: Actions in which some participants fail lead to inconsistency
Solution: Introduce a Coordinator that asks all Participants of a given Task if they can execute it (Prepare-Phase). The Task is only executed if all Participants can do so (Commit-Phase)
Benefits: Transparent/scalable solution, ensures atomicity of actions
Liabilities: Only covers predictable failures, Commit-Phase can still fail

Resource Lifecycle Manager



Intent: Decouple the resource usage from it's lifecycle management
Problem: Managing resource lifecycles is complex and error prone
Solution: Create one or more ResourceLifecycleMangers that can create, manage and release Resources. ResourceUsers can only acquire and release Resources using the RLM.
Benefits: Coordinates, centralizes and optimizes lifecycle management
Liabilities: Single Point of Failure, no flexibility for "special" resources

Microservice API

Structural Representation Patterns

Atomic Parameter

```
https://example.com/users/{userId}
```

Intent: Allow clients to send *one simple* parameter to the server
Solution: Define a name and value range for the parameter

Atomic Parameter List

```
https://example.com/stores?longitude=47.223&latitude=8.817
```

Intent: Allow clients to send *multiple simple* parameters to the server
Solution: Define a list of names and value ranges for the parameters

Atomic Parameter Tree

```
"profile": {    // Das ist ein Tree
  "age": 25,
  "phoneNumber": "076 123 45 67"
}
```

Intent: Allow clients to send *one complex* parameter to the server
Solution: Define a single root component that can contain tuples, arrays and other components (recursive). A tuple contains data with a specific name, type and value range.

Atomic Parameter Forest

```
{
  "profile": {    // Das ist ein Tree
    "age": 25,
    "phoneNumber": "076 123 45 67"
  },
  "activity": {   // Das hier ein anderer Tree
    "posts": [],
    "comments": []
  }
}
```

Intent: Allow clients to send *multiple complex* parameters to the server
Solution: Combine multiple *Parameter Trees* to one *Parameter Forest*

Pagination Pattern

Intent: Divide requests of large data into multiple parts ("pages")
Problem: Sending all data at once is inefficient if the data is large

Offset-Based Pagination

```
REQUEST: page_size: 50, page: 1
```

Solution: Use an offset (index) and a page size to return all data in the given range. Data must be sorted to ensure consistent access
Benefits: Easy to implement, good for data that doesn't change much
Liabilities: Data changes affect the results (duplicates, missing rows)

Cursor-Based Pagination

```
REQUEST: page_size: 50, page: "some-cursor"
```

Solution: Use a cursor (e.g. UUID) and a page size to return all data at that cursor. Data must be sorted to ensure consistent access
Benefits: Ensures more consistent results if the data changes
Liabilities: Navigation logic becomes more complex

Time-Based Pagination

```
REQUEST: page_size: 50, page: "2011-12-16 17:00:20"
```

Solution: Use a timestamp and a page size to return all data after that timestamp. Data must be sorted to ensure consistent access
Benefits: Ensures more consistent results if the data changes
Liabilities: Navigation logic becomes more complex

Version Identifier

```
GET /api/v2/users/123
```

```
GET /api/users/123
Host: v2.api.user-service.com
```

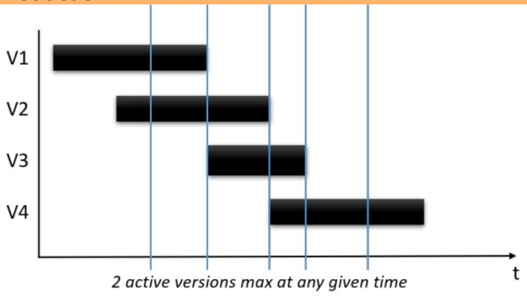
```
{
    "version": "2.0",
    "user": {
        username: "TEST"
    }
}
```

Intent: Indicate capabilities/incompatibilities of an API to clients
Solution: Introduce a *Version Identifier* in the description and exchange messages of an API to indicate its capabilities/incompatibilities

Semantic Versioning

Intent: Indicate the significance of changes between multiple versions
Solution: Use a three-number versioning scheme (`major.minor.patch`) to indicate breaking changes (`major`), compatible new functionalities (`minor`) and compatible bug fixes (`patch`). Both `minor` and `patch` versions may be hidden in the *Version Identifier* pattern if not needed

Two (N) in Production



Intent: Allow clients to migrate at their own speed to updated APIs
Solution: Provide two (or more) variants of the same API running on different/incompatible versions, adding/removing them gradually

Limited Lifetime Guarantee

Intent: Indicate to clients how long they can reliably use an API version
Solution: Define a fixed time-frame in which an API will be stable (non-breaking) and label each version with an expiration date

Game Programming Patterns

Game Loop

A UI that waits on user input until it progresses further (is blocking) is an application of the game loop pattern. **No**, Game Loop is non blocking.
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**, it only runs at a steady frame rate if a frame-cycle finishes before or equal to the fixed time.
The render() part can be skipped to keep the game time constant. **Yes**
Games logic always runs 60 times per second. **No**
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**
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**

Game Programming Component

In terms of maintainability in the future, is it a good idea to create a single monolithic game class? **No**, this approach leads to tightly coupled code, making it difficult to modify & understand it in the future.
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**
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**
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**
One benefit when using components is that the code base will contain fewer files/classes. **False**
Components always communicate with each other through the state of entities. **False**

Game Programming Event Queue

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**
Do event queues make it easier to take advantage of multi-threading? **Yes**
Do event queues allow you to make non-blocking calls? Yes
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
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**