1 Introduction

1.1 Pattern Definition

- Descriptions of successful engineering stories
- Address recurring problem
- Descripe generic solution that worked
- Tell about the forces of the problem (why is the problem hard)
- Tell about the engineering trade-offs to take (Benefits / Liabilities)
- Solution (Implementation)

1.2 Type of Patterns

- Architecture Patterns (Waiting Room)
- Software Patterns
  - Design Pattern (Elements of Reusable Object-Oriented Sofware)
  - Pattern-oriented Software Architecture (POSA)
- Organizational Patterns
- Learning and Teaching Patterns
- Documentation Patterns

1.3 Pattern Formats

1.3.1 POSA

- Pattern name
- Intent
- Problem
- Solution
- Benefits / Liabilities

1.3.2 Fault Tolerance

- Name
- Intent
- Solution
- Benefits / Liabilities

1.3.3 MAPI

- Name
- Intent
- Consequences

1.3.4 Game Programming Patterns

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

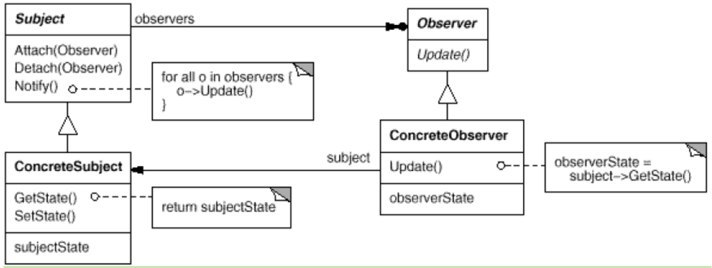
1.4 What are Patterns not?

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

2 GoF Patterns

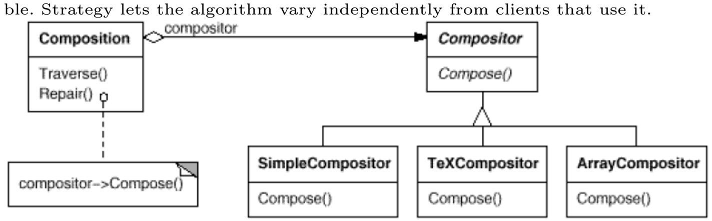
2.1 Observer

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



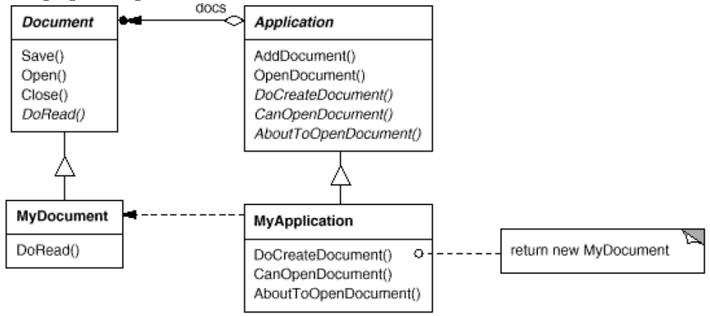
2.2 Strategy

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



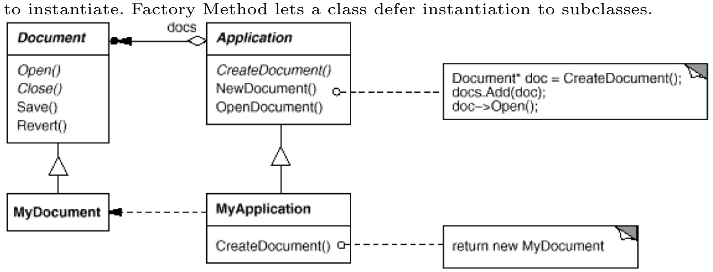
2.3 Template Method

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



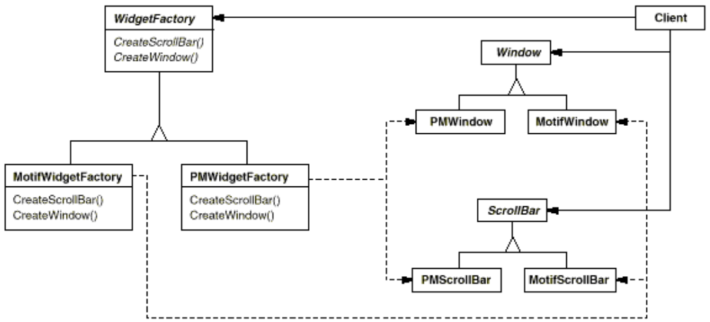
2.4 Factory Method

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



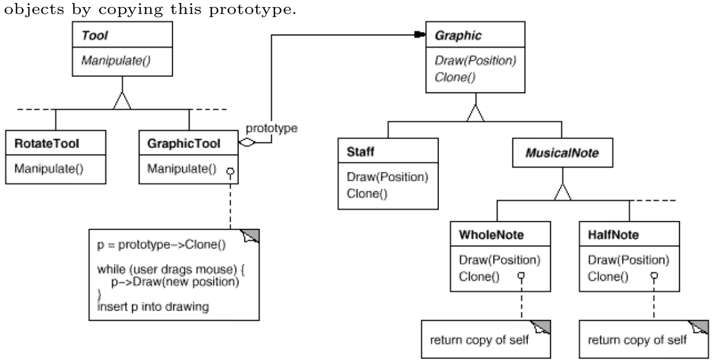
2.5 Abstract Factory

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



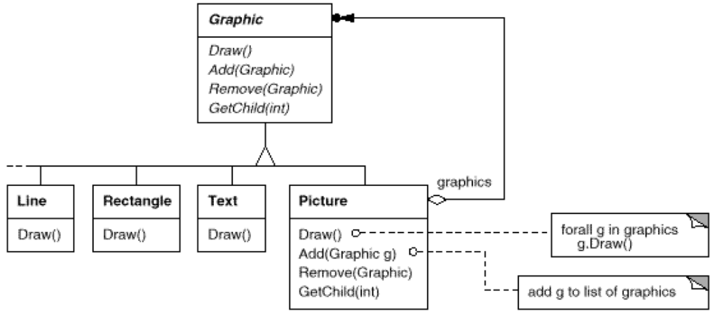
2.6 Prototype

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



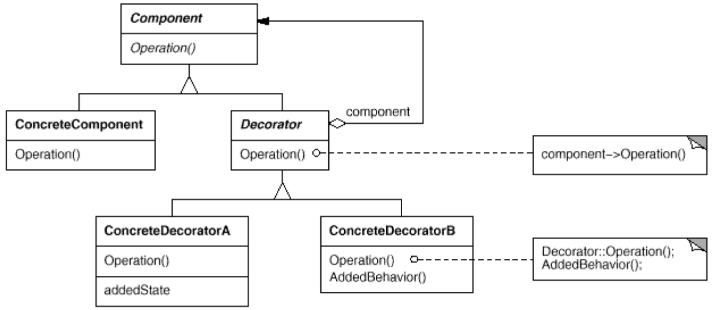
2.7 Composite

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



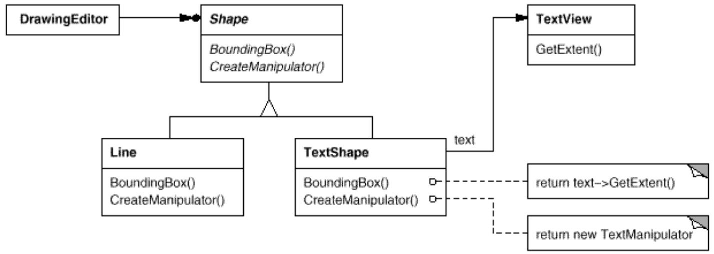
2.8 Decorator

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



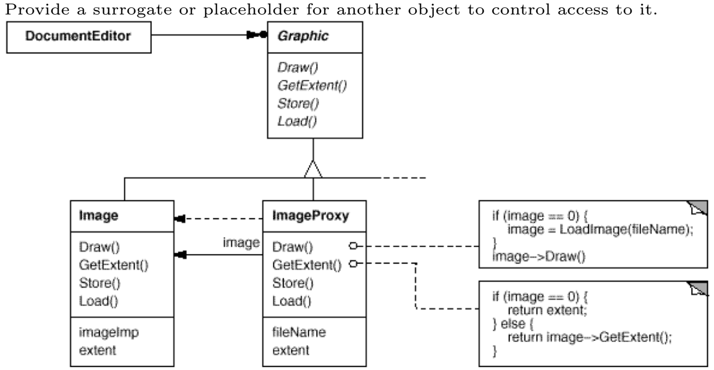
2.9 Adapter

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



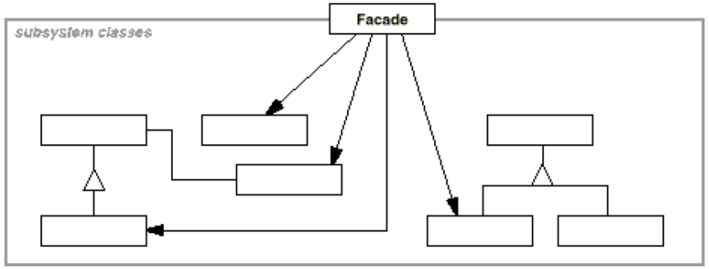
2.10 Proxy

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



2.11 Facade

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.



2.12 Mediator

2.12.1 Problem

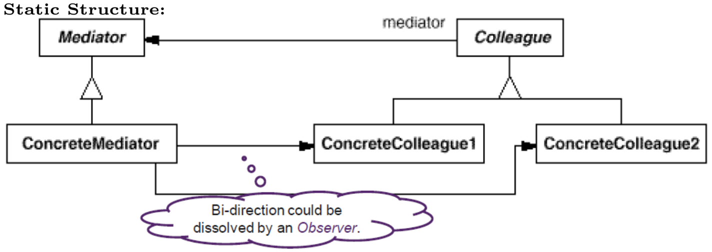
- Object Structures may result in many connections between objects
- In the worst case, every object ends up knowing about every other

Intent:

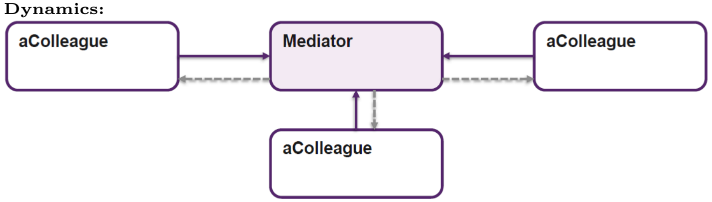
- How can strong coupling between multiple objects be avoided and communication simplified?

2.12.2 Solution

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



Dynamics:



2.12.3 Implementation

- Mediator as an Observer
- Colleagues act as Subject

Known Uses:

- Message Bus Systems
- Redux Dispatcher

2.12.4 Summary

Benefits:

- Colleague classes may become more reusable, low coupling
- Centralizes control of communication between objects
- Encapsulates protocols

Liabilities:

- Adds complexity
- Single point of failure
- Limits subclassing (of mediator class)
- May result in hard maintainable monoliths

2.13 Memento

2.13.1 Problem

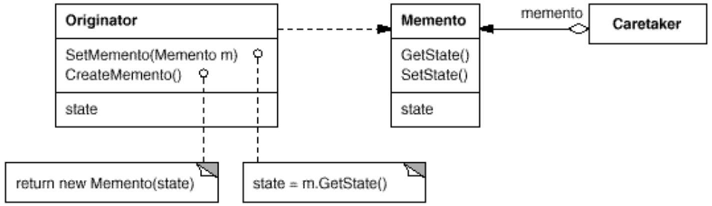
- Sometimes it's necessary to record the internal state of an object
- Objects normally encapsulate their state, making it inaccessible

Intent:

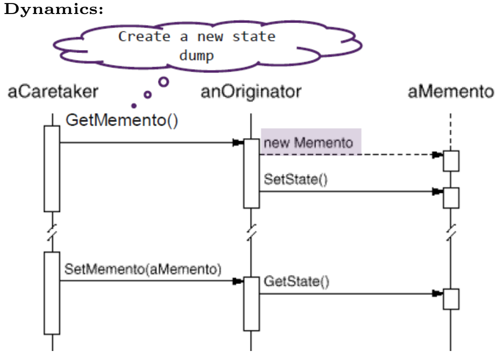
- How can the state of an object be externalized without violating its encapsulation?

2.13.2 Solution

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



Dynamics:



2.13.3 Participants

Memento

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

Originator

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

Caretaker

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

2.13.4 Implementation

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

2.13.5 Summary

Benefits

- Internal State of an object can be saved and restored at any time
- Encapsulation of attributes is not harmed
- State of objects can be restored later

Liabilities

- Creates a complete copy of the object every time, no diffs (memory usage)
- No direct access to saved state, it must be restored first

2.14 Command

2.14.1 Problem

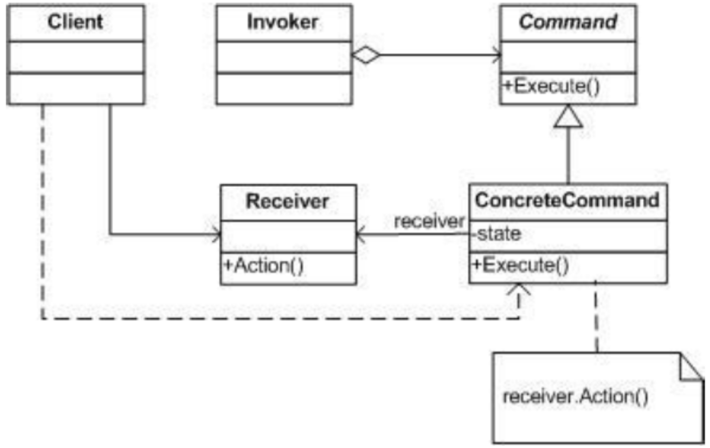
- Decouple the decision of what to execute from the decision of when to execute
- The execution needs an additional parametrization context

Intent:

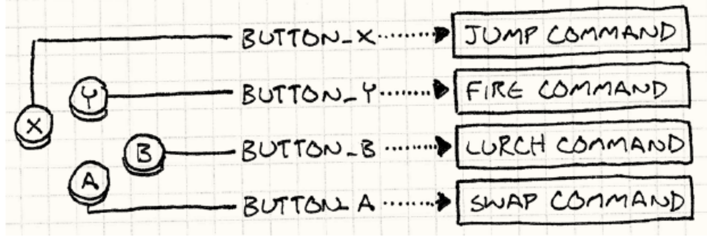
- How can commands be encapsulated, so that they can be parameterized, scheduled, logged and/or undone?

2.14.2 Solution

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



Dynamics:



2.14.3 Summary

Benefits:

- The same command can be activated from different objects
- New commands can be introduced quickly and easily
- Command objects can be saved in a command history
- Provides inversion of control, encourages decoupling in both time and space

Liabilities:

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

2.15 Command Processor

2.15.1 Problem

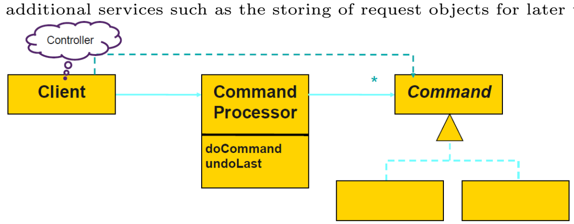
- Common UI applications support do and multiple undo steps
- Steps forward and backward are accessible in a history

Intent:

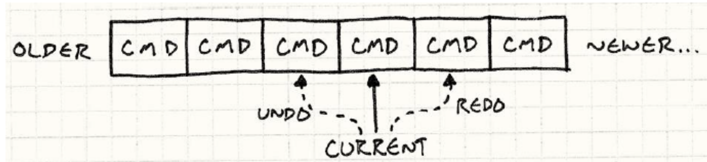
- How could we manage command objects, so the execution is seperated from the request and the execution can be undone later?

2.15.2 Solution

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



Dynamics:



2.15.3 Participants

Command Processor

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

Command

- A uniform interface to execute functions

Controller

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

2.15.4 Implementation

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

2.15.5 Summary

Benefits:

- Flexibility
- Allows addition of services related to command execution
- Enhances testability

Liabilities:

- Efficiency loss due additional indirection

2.16 Visitor

2.16.1 Problem

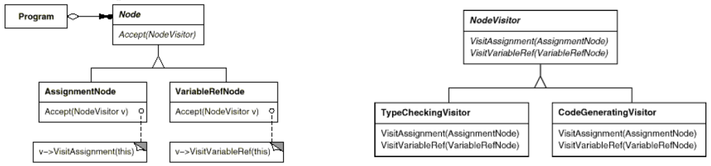
- Operations on specific classes needs to be changed/added without needing to modify these classes
- Different algorithms needed to process an object tree

Intent:

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

2.16.2 Solution

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



2.16.3 Implementation

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

Patterns that combine naturally with Vistor:

- Composite
- Interpreter
- Chain of Responsibility

2.16.4 Summary

Benefits:

- Visitor makes adding new operatios easy
- Separates related operations from unrelated ones

Liabilities:

- Adding new node classes is hard
- Visiting sequence fix defined within nodes
- Visitor breaks logic apart

3 Beyond GoF

3.1 External Iterator

3.1.1 Problem

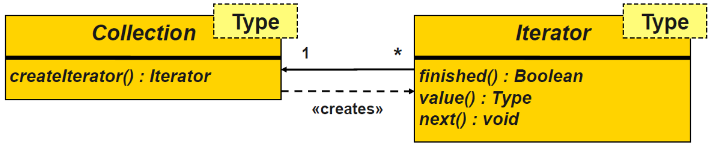
- Iteration through a collection depends on the target implementation
- Separate logic of iteration into an object to allow multiple iteration strategies

Intent:

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

3.1.2 Solution

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



Elementary operations of an Iterator's behaviour:

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

3.1.3 Summary

Benefits:

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

Liabilities:

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

3.2 Enumeration Method

3.2.1 Problem:

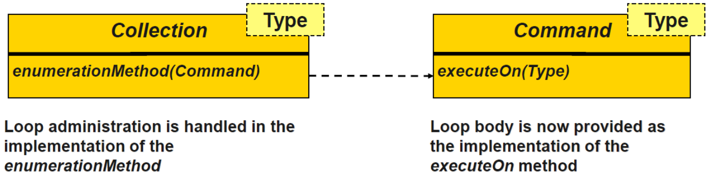
- Iteration management is performed by the collection's user
- Avoid state management between collection and iteration

Intent:

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

3.2.2 Solution

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



3.2.3 Summary

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

Benefits:

- Client is not responsible for loop housekeeping details
- Synchronization can be provided at the level of the whole traversal rather than for each element access

Liabilities:

- Functional approach, more complex syntax needed
- Often considered too abstract for programmers

3.3 Batch Method

3.3.1 Problem

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

Intent:

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

3.3.2 Solution

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

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

3.4 Objects for State

3.4.1 Problem

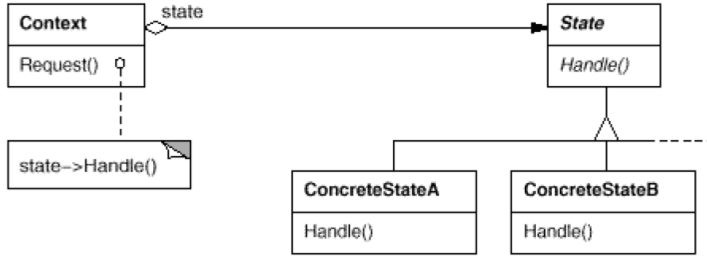
- Object's behaviour depends on its state, and it must change its behaviour at run-time
- Operations have large, multipart conditional statements (Flags) that depend on the state

Intent:

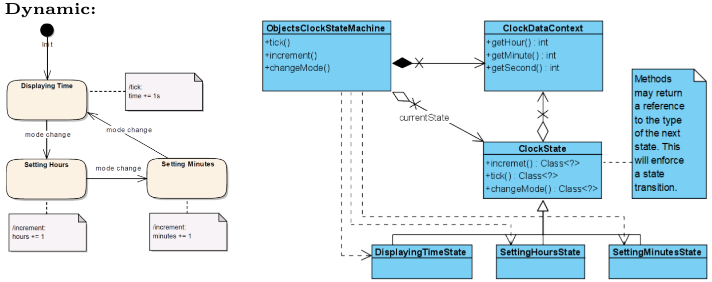
- How can an object act according to its state without multipart conditional statements?

3.4.2 Solution

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



Dynamic:



3.4.3 Summary

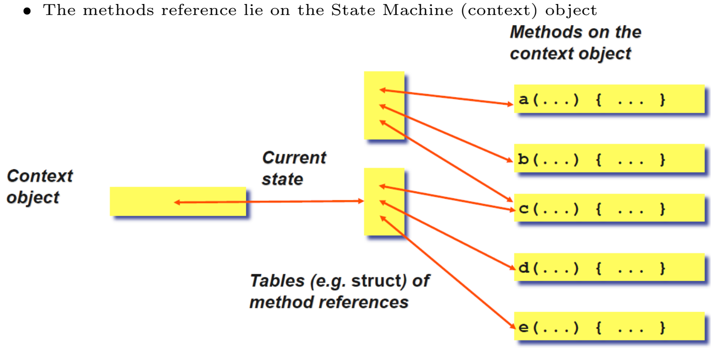
Criticism:

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

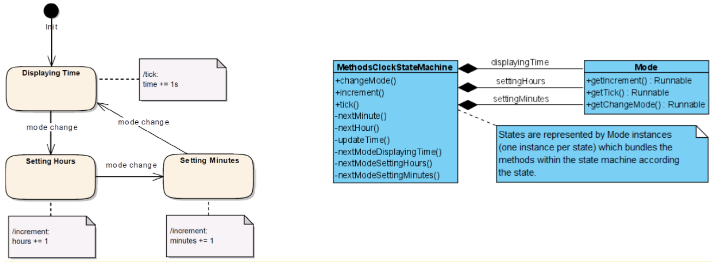
3.5 Methods for State

3.5.1 Solution

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



Dynamics:



3.5.2 Summary

Benefits

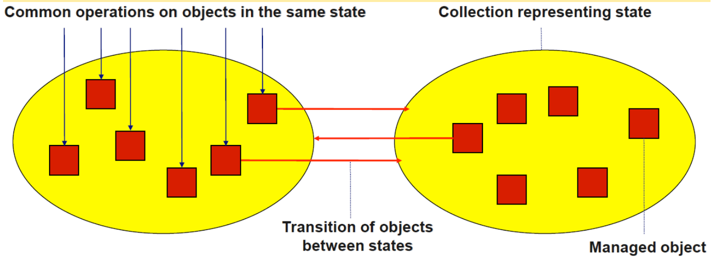
- Allows classes to express different behaviours in ordinary methods themself
- Behaviour coupled to the state machine, not scattered accross small classes
- Each distinct behaviour is assigned its own method

- No object context needs to be passed around, methods can already access the internal stte of the State Machine

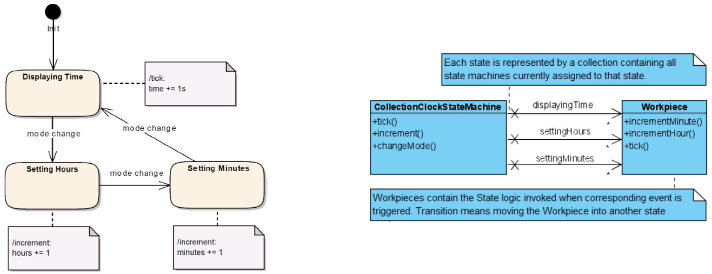
Liabilities
- Requires an additional two levels of indirection to resolve a method call
- The state machine may end up far longer than was intended or is manageable

3.6 Collection for State

3.6.1 Solution



Dynamics:



3.6.2 Summary

Benefits:
- No need to create a class per state
- Optimized for multiple objects (state machines) in a state
- Object's collection implicitly determines its state
- Can be compined with other state machine (Objects / Methods)

Liabilities:
- Can lead to a more complex state manager

3.7 Implementation of State Patterns

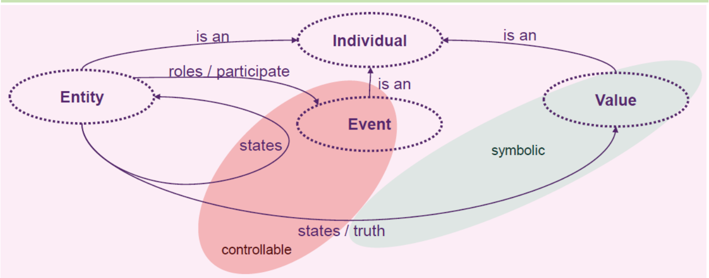
Objects:
- Results in a lot of classes and structures
- At least one class per state plus state machine

Methods:
- Propagates a single class with a lot of methods

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

4 System Analysis

4.1 Individuals



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

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

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

5 Software Design

5.1 Categories of Objects

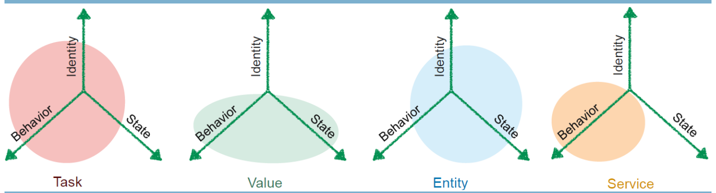
Entity
- Express system information
- Typically persistent in nature
- Identity is important to distinguish entity objects

Service
- Represent system activities
- Distinguished by their behaviour rather than their state

Values
- Interpreted content is the dominant characteristic
- Transient and do not have significant enduring indentity

Task
- Represent system activities
- Have an element of identity and state

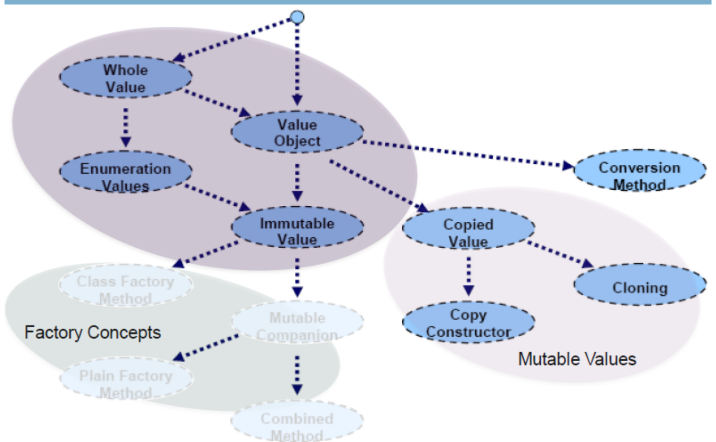
6 Object Aspects



7 Value Objects

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

8 Value patterns



8.1 Whole Value

8.1.1 Problem

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

8.1.2 Solution

Express the type of the quantity as a Value Class.

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

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

8.2 Value Object / Value Class

8.2.1 Problem

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

8.2.2 Solution

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

- Override Object's methods who define equality
- Java
  - *equals(Object other)*
  - *hashCode()*
  - implement *Serializable / toString()* if appropriate
- TypeScript
  - *equals(other: Object)*
  - implement *toString()* if appropriate
- Overriding *toString()* can help using your Value Object

8.3 Conversion Method

8.3.1 Problem

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

8.3.2 Solution

Provide type conversion methods responsible for converting Value Objects into related formats.
- Provide a constructor which converts between types *String(char[] value)*
- Or create a conversion instance method that converts to other type *Date.toOtherType()*
- Or create a **Class Factory Method** with conversion characteristics *Date.from(Instant i)*

8.4 Immutable Value

8.4.1 Problem

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

8.4.2 Solution

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

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

8.5 Enumeration Values

8.5.1 Problem

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

8.5.2 Solution

Treat each constant as Whole Value instance declaring it public.
- Implement a Whole Value and declare the Enumeration Values as *public readonly* fields
- Prevent inadvertently changing the constants
- Pattern is built in (enum)

8.6 Copied Value and Cloning

8.6.1 Problem

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

8.6.2 Solution

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

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

8.7 Copy Constructor

8.7.1 Problem

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

8.7.2 Solution

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

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

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

8.8 Class Factory Method / Simple Factory

8.8.1 Problem

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

8.8.2 Solution

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

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

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

8.9 Mutable Companion

8.9.1 Problem

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

8.9.2 Solution

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

- Factory Object for immutable values
- Neither a subtype nor a supertype of the Immutable Value

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

8.10 Relative Values

8.10.1 Problem

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

8.10.2 Solution

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

- Override-Overload Method Pair: (*compareTo(T other)* and *equals(T other)*)
- Bridge Method: Provide a Method for *equals(Object other)* and forward it to *equals(T other)*

-

8.11 Discussion

Difference between Whole Value and Immutable Value

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

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

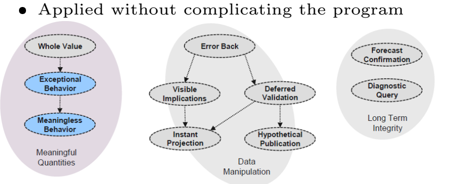
- Factory: A foreign value type should be converted into the current value format
- Contructor: A more generic type should be converted into the current type

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

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

9 Checks

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



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

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

9.1 Exceptional Behaviour

9.1.1 Problem

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

9.1.2 Solution

Use one or more distinguished values to represent exceptional circumstances.

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

```
// TypeScript
public div(num: number, div: number): number | CalcError {
    div === 0 ? CalcError.DivByZero : num / div;
}
```

9.2 Meaningless Behaviour

9.2.1 Problem

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

9.2.2 Solution

Write methods with minimalistic concern for possible failure.

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

```
// TypeScript
public div(num: number, div: number): number {
    return num / div; // Infinity (Java NaN) if div == 0
}
```

10 Framework Introduction

Why Frameworks

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

What is a Framework

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

Framework Callbacks

- Hollywood Principle: Don't call us, we call you!
- Extendability and configurability

10.1 Application Framework

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

10.2 Examples

Frameworks

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

Application Framework

- Spring
- ASP.NET
- Angular

10.3 Difference: Library / Framework / App Framework

Library

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

Framework

- Provide Hooks / Extension points
- Strogly rely on Inversion of Control (IoC)
- Defines when hooks are called, thus controlls part of the app flow

App Framework

- Contains the *main()* procedure
- Completely controlls the app flow

10.4 Summary

Benefits

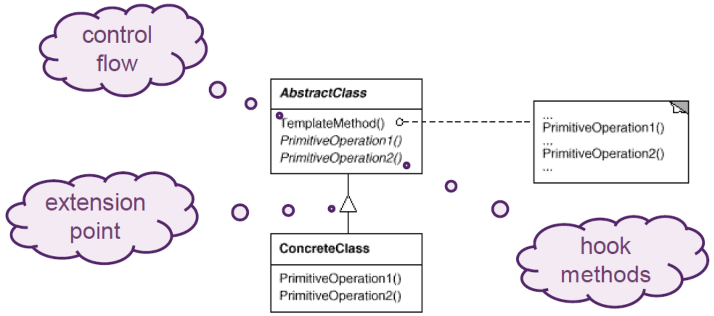
- Less code to write
- Reliable and robust code
- Consistent and modular code
- Reusability
- Maintenance

Liabilities

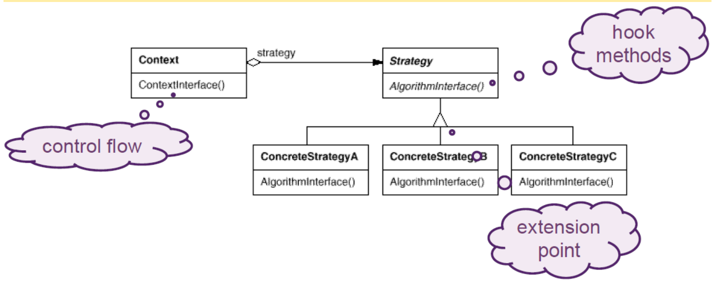
- Portability: Code is strongly coupled to the overlying Framework
- Testing: Close coupling between framework parts
- Evolution: User's impletation may break due next Framework version

10.5 Hook / Extension Point / Control Flow

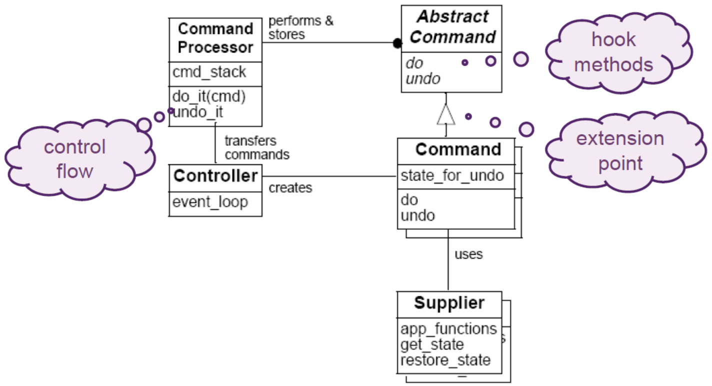
10.5.1 Template Method



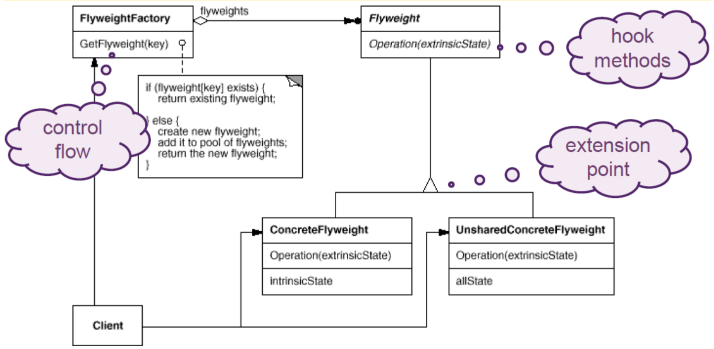
10.5.2 Strategy



10.5.3 Command Processor



10.5.4 Flyweight

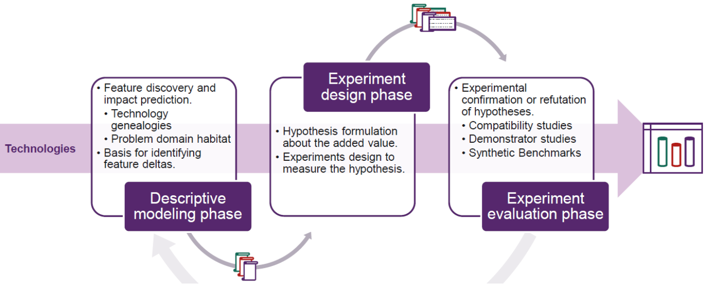


11 Meta Frameworks

A Framework for Evaluating Software Technology

- Initial acquisition cost
- Long-term effect
- Training and support
- Future technoloy plans
- Response of direct competitor organizations

11.1 Framework Evaluation Phases



12 Developing Frameworks

- Frameworks need evolutionary improvements

12.0.1 Frameworkers Dilemma

Potential ways out of the dilemma

1. Think very hard up-front
2. Don't care too much about framework users
3. Let framework users participate
4. Use helping technology

13 Meta Pattern

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

Recurring Problems

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

13.1 Reflection

Usage (Java / C#)

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

Provides Facility to implement:

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

Consists of two aspects:

Introspection

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

Intercession

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

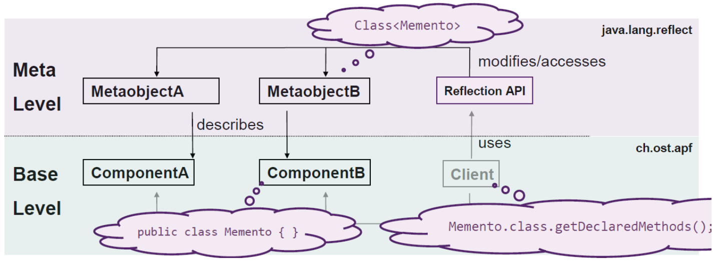
Defines meta level and base level:

Meta level

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

Base level

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



13.1.1 Summary

Benefits

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

Liabilities

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

13.2 Meta Objects

Usage

- Classes
- Object Attributes
- Methods
- Class Relationships

13.3 Type Object

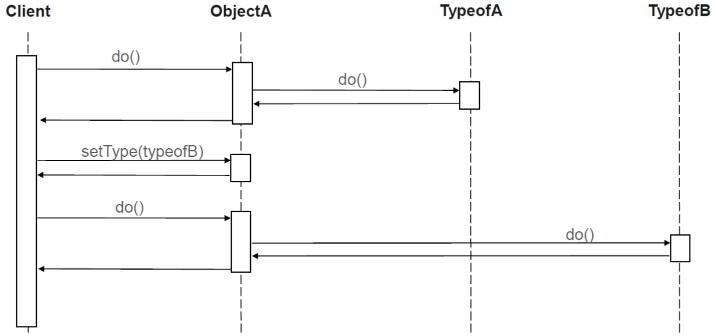
13.3.1 Problem

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

13.3.2 Solution

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

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



13.3.3 Summary

Benefits

- Categories can be added easily, event at runtime
- Allows multiple meta-levels

Liabilities

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

13.3.4 Discussion

Based on which GoF Pattern

- Strategy

Similar intent in a GoF Pattern

- State, also changes at runtime

13.4 Property List

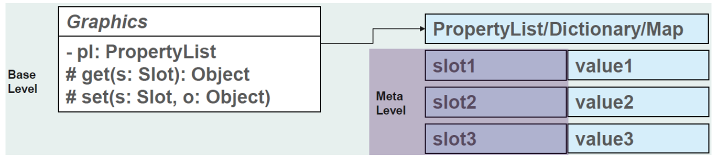
13.4.1 Problem

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

13.4.2 Solution

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

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



13.4.3 Summary

Benefits

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

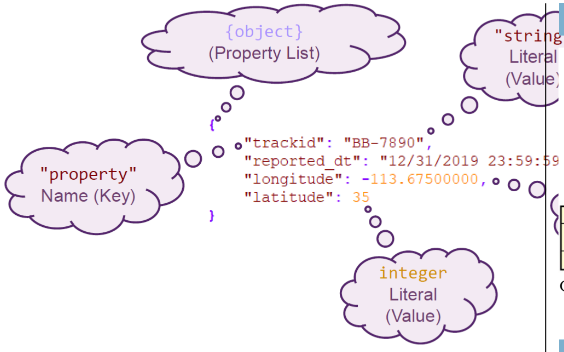
Liabilities

- Different ways to access regular / dynamic attributes
- Type safety left to the programmer
- Run-time overhead
- Memory Management

Mitigate Liabilities: Bridge Methods

13.5 Anything

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



Object Definition:

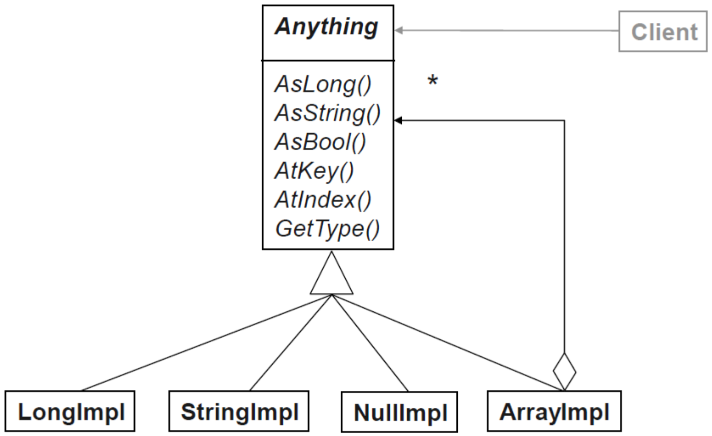
13.5.1 Problem

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

13.5.2 Solution

Create an abstraction for structured values that is self describing.

- Implement a representation of simple values
- Add an implementation for a sequence of values (& key value access)
- Provide a default value



13.5.3 Summary

Benefits

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

Liabilities

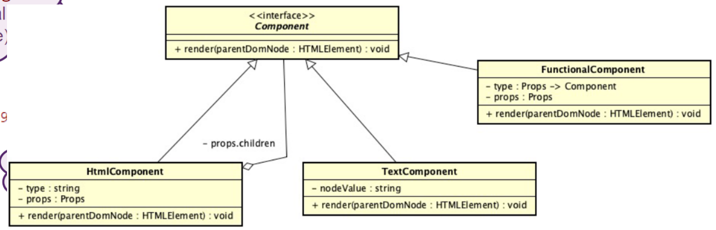
- Less type safety
- Intent of parameter elements not always obvious
- Overhead for value lookup and member access
- No real object, just data

13.5.4 Discussion

Which GoF Pattern does Anything implement?

- Transparent Composite Pattern
- Null Object

14 Reimplementing React



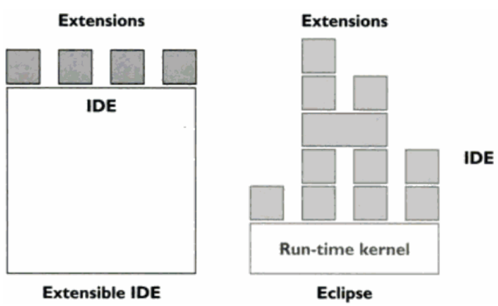
Code????

15 Reimplementing Redux

Code???

16 Eclipse

16.1 Components



16.1.1 SWT

- Native Widgets and mostly written in Java
- No performance implact form UI
- Provides basic Components

16.1.2 JFace

- Builds on top of SWT
- Actions
- Menus
- Dialogs
- Wizards
- Fonts

16.1.3 OSGi

- Specifies a component and service model for Java
- Components can be installed, started, stopped and uninstalled at runtime
- Each bundle gets its own classloader
- Bundles define dependencies and export some of their packages

Bundles:

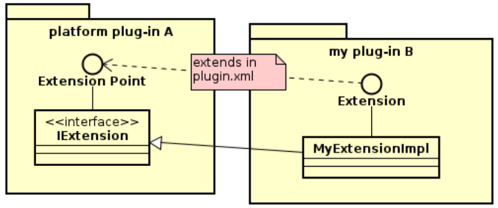
- Eclipse Plugins are OSGi Bundles
- Packaged as JARs
- MANIFEST.MF contains the bundle metadata

Services:

- Connect Bundles
- Service is a POJO and can be registered at a Service Registry at bundle start
- Eclipse does not use OSGi services but Extensions and Extension Points

16.1.4 Extension Points

- Collect contributions to the plug-in offering the Extension Point
- Each plug-in contributes to at least one extension point
- EP and Extensions are not part of the Manifest but separate XML files



16.2 Eclipse Plug-ins

- Add new Actions, Editors, Views, Perspectives, Preferences
- Each one gets its own class loader
- Can only access specified dependencies
- Has three interesting files
  - Manifest
  - plugin.xml: wires up the Extension
  - Class that imlements the Extension

Eclipse Platform

- Manages Plugins
- Discover installed plug-ins
- builds the extension registry
- connects extesions and EP
- Only activates plug-ins when needed

16.3 GoF Design Patterns in Eclipse

Platform
- Singleton: getting Workbench, Plug-in

Workspace Resources
- Proxy and Bridge: Accessing File System
- Composite: the workspace
- Observer: tracking resource changes

Core Runtime
- IAdaptable and Adapter Factories: Property View SWT

SWT
- Composite: composing widgets
- Strategy: defining layouts

JFace
- Observer: responding to events
- Pluggable adapter: Connecting widget to model
- Strategy: customize a viewer without subclassing
- Command: Actions

UI Workbench
- Virtual Proxy: lazy loading with E.P.

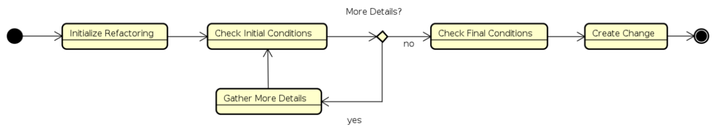
LTK
- Template Method, Composite, Memento

CDT
- Visitor: to traverse the AST

16.4 Refactoring Language Toolkit (LTK)

- Language neutral API for refactorings
- Classes for constructing refactoring UI (common L&F for wizards)
- Refactoring wizard with diff preview
- Classes for modelling resource changes
- Refactoring participant functionality

16.4.1 Refactoring Lifecycle Overview



16.4.2 Refactoring Participants

- Can participate in the condition checking and change creation of a refactoring processor
- Refactorings that change multiple files may have an impact on other tools

17 Spring Framework

Provides a programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform.

17.1 Java Annotations

- Metadata that do not directly affect the code they annotate
- Uses
  - Information for the compiler
  - Compile-time and deployment-time proccessing
  - Runtime processing

17.1.1 Examples

Information for the compiler

- @Override
- @Deprecated
- @SuppressWarnings
- @FunctionalInterface

Compile-time and deployment-time processing

- @NonNull
- @Data and other Lombok annotations

Runtime processing

- @Test, @Before, @After and other Junit annotations
- @RestController, @GetMapping, @PathVariable and other Spring annotations

17.1.2 Declaring Annotations

TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE, ANNOTATION_TYPE, PACKAGE, TYPE_PARAMETER, TYPE_USE, MODULE

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation {
    int value();
    String optional() default "";
}
```

Annotation "methods" must be assigned when used, unless default value is set

SOURCE: discarded by the compiler
CLASS: in .class, ignored by runtime
RUNTIME: in .class, available at runtime

```java
@MyAnnotation(value = 42)
public class MyClass
```

17.1.3 Processing Annotations

- Depending on retention policy
  - processed by compiler
  - during compilation by Annotation Processors
  - at runtime using reflection
- Processors need to be on the classpath to be recognized by the Java compiler
- Compiler then invokes the processor if it finds any annotations the processor has registerd
- Processors can generate new files, those can contain additional annotations

17.2 Core Principles

- DI
- IoC
- Aspect Oriented Programming (AOP)

17.3 Beans

- Form the backbone of the application
- Managed by the Spring IoC Container

17.3.1 Scopes

Control the lifetime of Beans

- *singleton*: (default) single object for each Spring IoC Container
- *prototype*: Any number of instances
- *request*: Lifecycle of a single HTTP request, each has its own instance
- *session*: Lifecycle of a HTTP Session
- *application*: Lifecycle of a ServletContext
- *websocket*: Lifecycle of a WebSocket

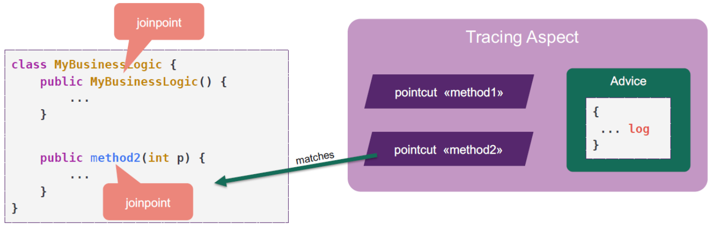
```java
@Bean
@Scope('singleton')
public DataSource dataSource() { }
```

17.3.2 Config Simplifications

@Component: Spring creates beans automatically
@ComponentScan: Automatic scanning for components
@Autowired: Property is mapped to Constructor

17.3.3 Aspect Oriented Programming (AOP)

- Mitigates cross-cutting concerns



- When a *pointcut* pattern matches a *joinpoint* being executed, an advice can be run
- Weaving is the compile or runtime technology to interleave the advice code into the joinpoints

17.3.4 AspectJ

- Technology of Spring AOP
- Can be used on all objects
- Offers different approaches to weave aspects:
  - Compile-time weaving (produces woven class files as output)
  - Post-Compile weaving (weaves existing class files)
  - Load-time weaving (binary weaving)

18 Boxing / Killing

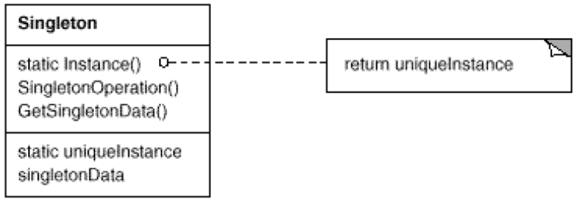
18.1 Singleton Boxing / Killing

18.1.1 Problem

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

18.1.2 Solution

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



18.1.3 Solutions inside Singleton

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

18.1.4 Summary

Benefits

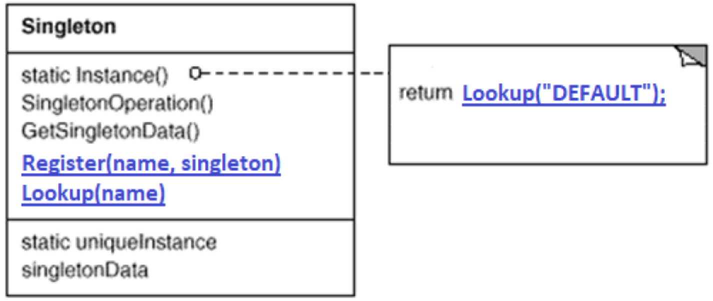
- Controlled access to sole instance
- Reduced name space
- Permits refinement of operations and representation
- Permits variable number of instances
- More flexible than class operations

Liabilities

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

18.2 Singleton Variation 'Registry'

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



18.3 Monostate (Borg) - Killing

18.3.1 Problem

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

18.3.2 Solution

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

```java
public class Monostate {
    private static int x;
    public int getX() { return x; }
}
```

18.3.3 Summary

Benefits

- Transparency, no need to know about Monostate
- Derivability
- Polymorphism
- Well-defined creation and destruction

Liabilities

- Breaks inheritance hierarchy
- Memory usage
- Unable to share Monostate across several tiers

18.4 Service Locator

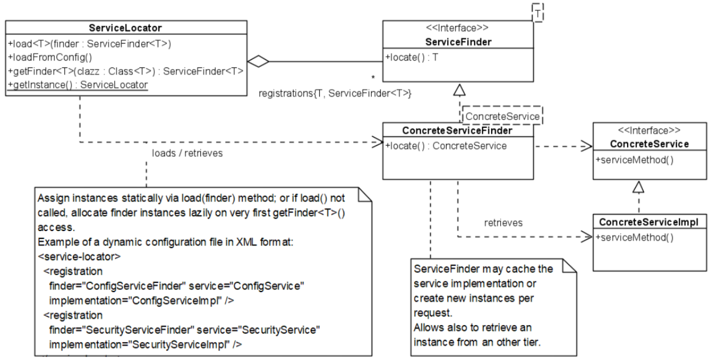
18.4.1 Problem

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

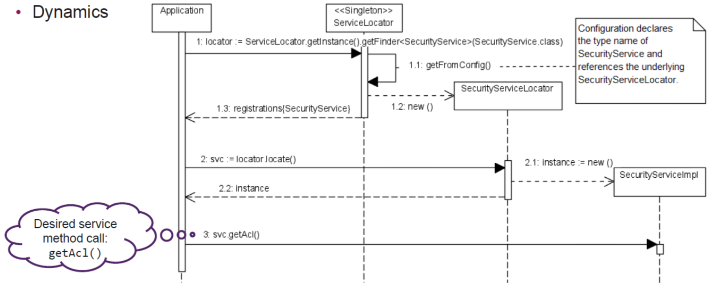
18.4.2 Solution

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

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



- Dynamics



18.4.3 Summary

Benefits

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

Liabilities

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

18.5 Parameterize from Above

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

18.5.1 Solution

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

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

18.5.2 Summary

Benefits

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

Liabilities

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

18.6 Dependency Injection

18.6.1 Problem

- User may override implementations of existing app components (e.g. test doubles)
- Any Componet within the system can demand an object of a specified interface
- The Components should not know anything about the wiring mechanism

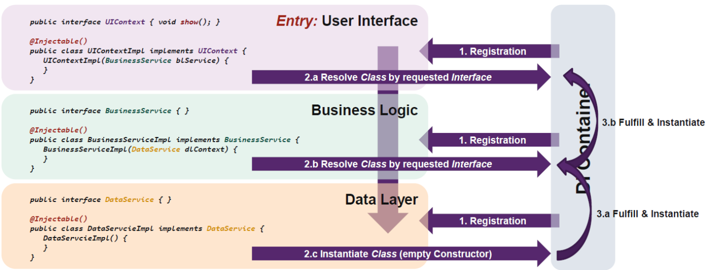
18.6.2 Solution

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

- Central container
- Users reference dependencies by the required interfaces
- Users apply code annotations
- Clients should not address the container directly

18.6.3 Implementation

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



18.6.4 Summary

Benefits

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

Liabilities

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

18.6.5 Discussion

Relation Singleton - DI

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

Improvement of DI over Service Locator

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

18.7 Flyweight

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

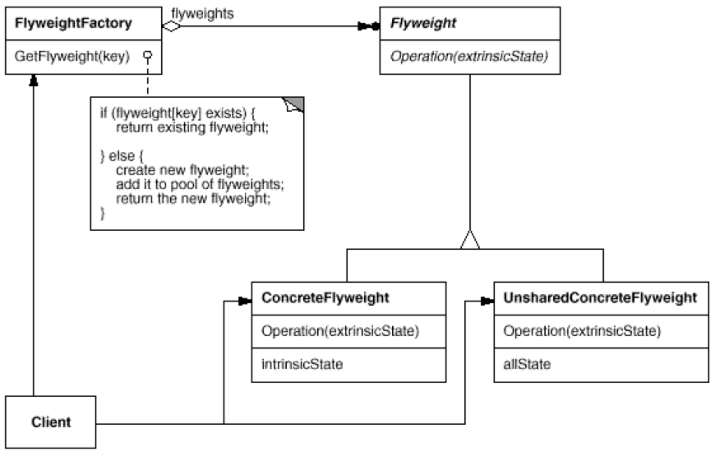
18.7.1 Problem

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

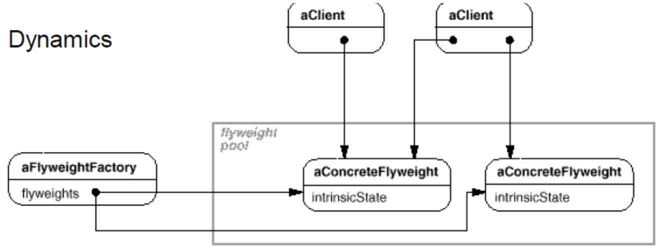
18.7.2 Solution

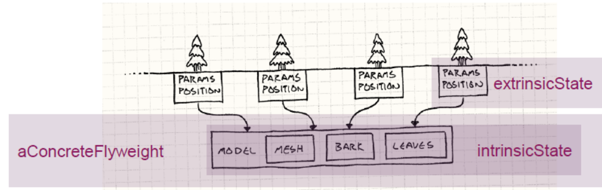
Use sharing to support large numbers of fine-grained objects efficiently

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



Dynamics





18.7.3 Solutions inside Flyweight

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

18.7.4 Summary

Benefits

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

Liabilities

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

18.8 Pooling (Boxing)

18.8.1 Problem

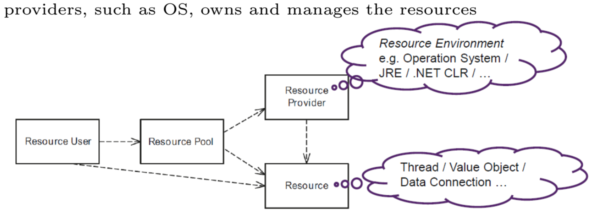
- A fast and predictable access to resources should be provided
- Wastage of CPU cycles in repetitious acquisition / release should be avoided
- Acquisition / release complexity should be minimized
- How can expensive acquisition and release of resources be avoided by recycling resources that are no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

18.8.2 Solution

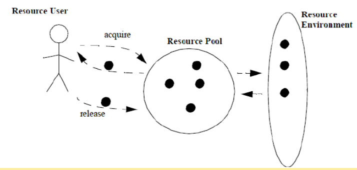
Manage multiple instances of one type of resource in a pool. This pool of resources allows for reuse of released resources.

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

Static Structure



Dynamics



18.8.3 Implementation

- Define the maximum number of resources that are maintained by the pool
- Decide between *eager* and *lazy* Acquisition
- Determine resources recycling/eviction semantics

18.8.4 Summary

Benefits

- Performance of app

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

Liabilities

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

18.8.5 Discussion

Pattern that can be combined with Pooling

- Pool acts as mediator

Relation of Flyweight and Pooling

- Flyweight implements a pool with immutable resources statically

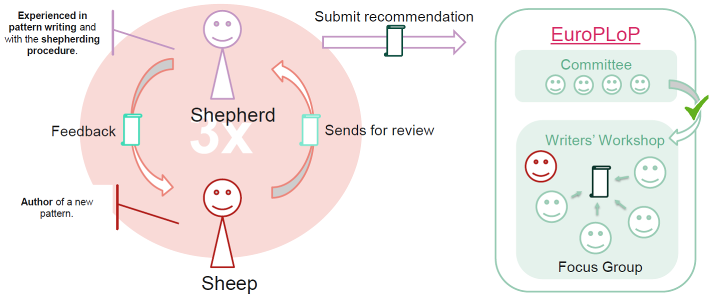
Is immutability of named resources key?

- No

Difference between pooling and caching

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

19 Shepherding Process



19.1 Process

1. Three Iterations

- How to budget your time and effort to make shepherding effective

2. The Shepheade Know the Sheep

- How to establish a productive relationship between you and the author

3. Half a Loaf

- How to make sure that shepheadring continous to move forward

4. Big Picture

- How to gasp the gist of the pattern right off the bat

5. Author as Owner

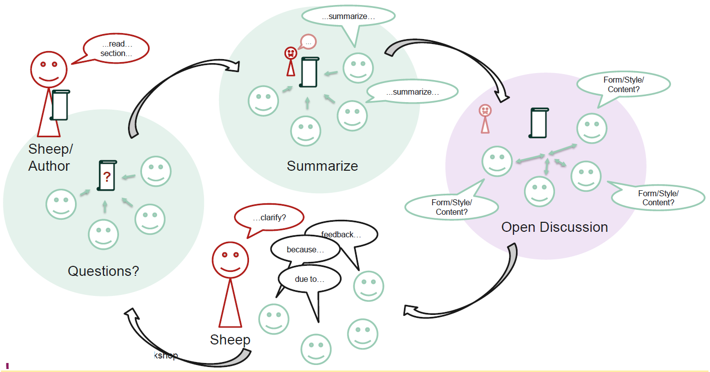
- How to keep from writing the pattern for the author

6. Forces Define Problem

- How to understand the problem at a deeper level

19.2 Writer's Workshop

- Used to improve patterns
- Primary focus of PLoP (PATTERN LANGUAGES OF PROGRAMS)
- The authors of the paper under discussion remain silent
- Major target: Get a lot of feedback and suggestions



19.2.1 1) Pattern Scanning

Does reading the pattern problem, solution, known uses, context, forces, consequences make sense?

19.2.2 2) Styling the Forces

Are forces listed as items?

19.2.3 3) BUT Style

- Is BUT-Style used and does it build tension?
- Does it lead to bold face solution?

19.2.4 4) Detailed Example / Technical Diagram

Is there a detailed example and technical diagram?

19.2.5 5) Known Uses

Are there at least three know and appropriate uses

19.2.6 6) Relationship

- Are the related patterns described in a logical order?
- Are the relationships appropriate?

19.2.7 7) Sstand-Alone, Self-Contained

- Does the pattern overlap with other patterns?
- Is the pattern description coherent?
- Are there suggestions for improvement?