Applying the Singleton pattern

- Declare class *UserSession* to be a singleton.
- Instantiation of *instance* is named *createUserSession*.
- Extend implementation of *createUserSession* by further case distinctions (number of user sessions is smaller than allowed maximum, successful authentication).

Facade I

Classification object/structural

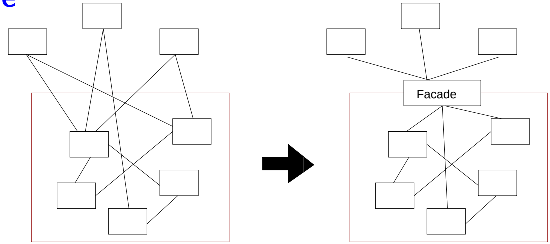
Intent Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Applicability Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem. A facade can provide a simple default view of the subsystem that is good enough for most clients.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Facade II

Structure



Participants

- *Facade*
  - knows which subsystem classes are responsible for a request
  - delegates client requests to appropriate subsystem objects
- subsystem classes
  - implement subsystem functionality
  - handle work assigned by the facade object
  - have no knowledge of the facade, i.e. no reference to it

Facade III

Consequences The facade

- shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- promotes weak coupling between subsystems and clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.
- doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

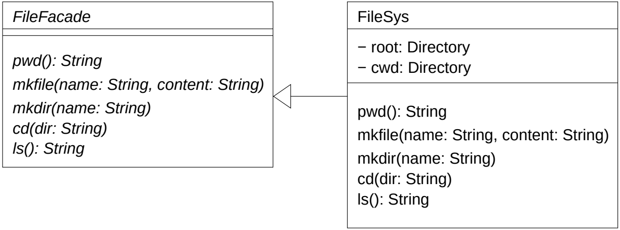
Related Patterns Abstract Factory, Mediator

Example: file system

Uniform interface for file system:

- File system API contains different classes, whose interaction is difficult to understand.
- In particular, the admissible consequences for generating file structures are not clear.
- In real file systems: uniform interfaces for handling the different phenomena file, directory, alias, ...

Applying the Facade pattern



- Implementing *FileSys* of *FileFacade* contains two private attributes
- Creation routine generates a *root*-directory “/” and sets *cwd* to *root*
- *pwd* calls *cwd.getName*
- *mkfile* calls *cwd.add*
- . . .

Proxy I

Classification object/structural

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

Also Known As Surrogate

Applicability Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

Some situations in which the Proxy pattern is applicable:

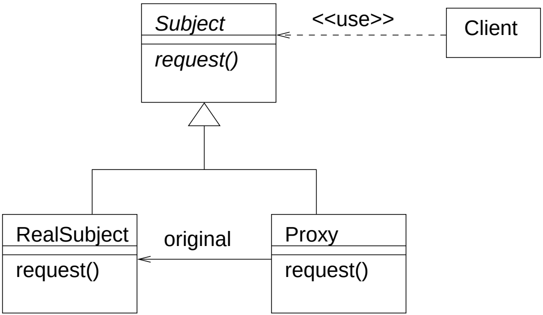
1. A *remote proxy* provides a local representative for an object in a different address space.
2. A *virtual proxy* creates expensive objects on demand (delayed loading, delayed generation).

Proxy II

3. A *protection proxy* controls access to the original object. Protection proxies are useful when objects should have different access rights.
4. A *smart reference* is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - counting the number or references to the real object so that it can be freed automatically when there are no more references (also called *smart pointer*)
   - loading a persistent object into memory when it's first referenced
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Proxy III

Structure



Proxy IV

Participants

- *Proxy*
  - maintains a reference that lets the proxy access the real subject
  - provides an interface identical to *Subject*'s so that a proxy can be substituted for the real subject
  - controls access to the real subject and may be responsible for creating and deleting it
- *Subject*
  - defines common interfaces for *RealSubject* and *Proxy*, so that the proxy can be used anywhere a real subject is expected
- *RealSubject*
  - defines the real object that the proxy represents

Related Patterns Adapter, Decorator

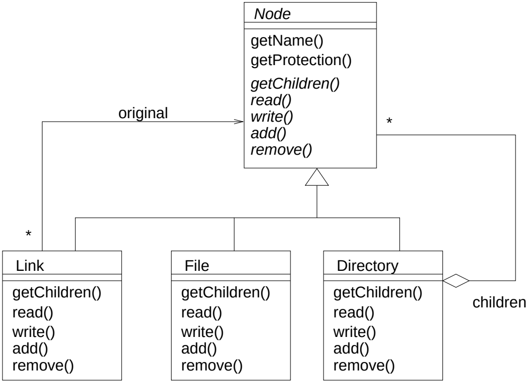
Example: file system

Introduction of aliases

- file alias
  - “symbolic link” in Unix
  - “alias” in MacOS
  - “shortcut” in Windows95+
- operations on files and aliases
  - alias permits all operations that are possible on originals
  - forwards operations to the original
  - special interpretations of operations is possible in special cases (e.g., for copying)

Applying the Proxy pattern to the file system

New class *Link* as proxy for *Node*



Client-Dispatcher-Server I

Classification structural

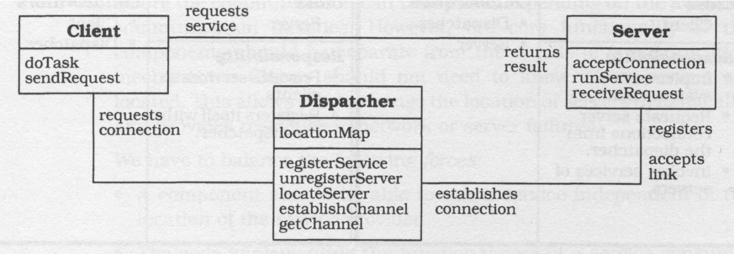
Intent/Problem Buschmann et al. (1996)

- Software system uses servers distributed over a network
- Connection between components have to be established before communication
- Core functionality should be separated from communication details
- Clients should not need to know where servers are located

Also Known As -

Motivation/Applicability Services are located on different servers

Structure

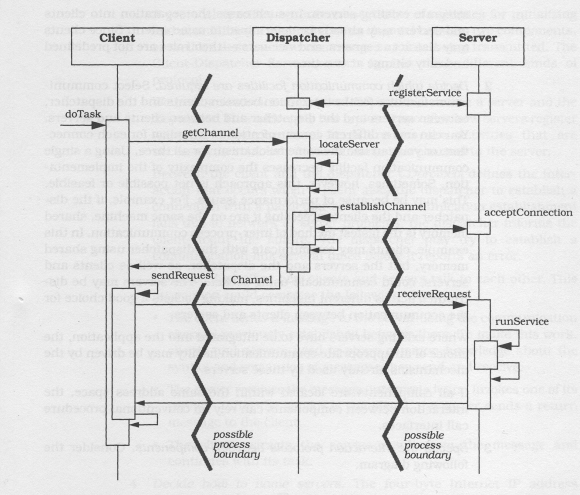


Client-Dispatcher-Server III

Participants/Consequences/Implementation

- Provide a **dispatcher** to act as an intermediate layer between client and server
- Dispatcher implements a name service to provide location transparency
- Dispatcher establishes the communication
- **Servers** provide services to other components
- Servers have unique names and are connected to the dispatcher
- **Clients** rely on the dispatcher to locate a particular service and to establish a connection

Dynamics


Client
Dispatcher
Server
registerService
doTask
getChannel
locateServer
establishChannel
acceptConnection
sendRequest
Channel
receiveRequest
runService
possible process boundary
possible process boundary

Client-Dispatcher-Server V

Sample Code see Buschmann et al. (1996)
Known Uses RPCs, CORBA
Related Acceptor and Connector

Forwarder-Receiver (Peer-to-peer) I

Classification structural
Intent/Problem Buschmann et al. (1996)

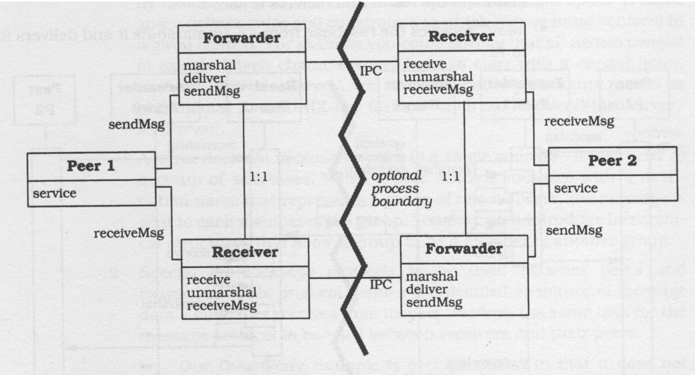
- Commonly distributed applications use efficient low-level mechanisms for inter-process communication (e.g., TCP/IP, message queues)
- Low-level mechanisms often introduce dependencies on the underlying operating system and network protocol, which restricts portability
- Higher-level mechanisms like remote procedure calls are less efficient
- Communication mechanism should be exchangeable
- The senders should only need to know the names of their receivers
- The communication should not have major impact on performance

Forwarder-Receiver (Peer-to-peer) II

Also Known As Peer-to-peer

Motivation/Applicability Efficient communication between peers

Structure

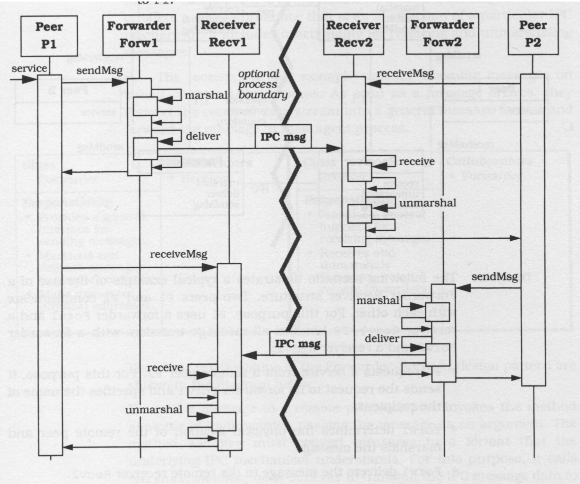


Forwarder-Receiver (Peer-to-peer) III

Participants/Consequences/Implementation

- Distributed **peers** collaborate to solve a particular problem.
- A peer may act as a client, a server, or both.
- The details of the underlying communication mechanism are hidden from peers
- System-specific functionality (name mapping to physical locations, communication channel establishment, marshaling) is encapsulated into separate components.
- A **forwarder** marshals the data and sends messages to other peers
- A **receiver** receives and unmarshals the data.

Dynamics


Peer P1
Forwarder Forw1
Receiver Recv1
Receiver Recv2
Forwarder Forw2
Peer P2
service
sendMsg
optional process boundary
receiveMsg
marshal
deliver
IPC msg
receive
unmarshal
receiveMsg
sendMsg
marshal
IPC msg
deliver
receive
unmarshal

What have we learned on design patterns? I

- Design patterns are object-oriented patterns at detailed design level.
- They are closer to implementation than architectural styles.
- According to the classification of Gamma et al. (1995), there are behavioral, creational and structural patterns.
- Design patterns support achieving desirable properties in implementing object-oriented software, e.g. independent modification of parts, limitation of communication paths etc.

What have we learned on design patterns? II

- We have presented and used the following patterns for MVC:
  1. Composite
  2. Observer
  3. Strategy
  4. Factory Method

  plus the patterns

  5. Singleton
  6. Facade
  7. Proxy
  8. Client-dispatcher-server
  9. Forwarder-Receiver

Idioms

Characteristics

- Specific patterns for (object-oriented) programming languages
- Low abstraction level
- Describe, how certain aspects of components or relations between components can be implemented by means of a specific programming language

Literature

- Buschmann et al. (1996)
- Coplien (1992)
- Coplien (1998)
  http://users.rcn.com/jcoplien/Patterns/C++Idioms/EuroPLoP98.html

Application

- Solution of implementation-specific problems in a certain programming language, e.g.
  - memory management
  - creation of objects
- Implementation of design patterns
- Description of programming styles, e.g.
  - names for operations
  - formatting of source code
- Simplified communication between developers

Idiom for implementing "Singleton" in C++ I

Name Singleton (C++)

Problem An implementation of the Singleton design pattern is needed to ensure that only one instance of a class exists at runtime.

Solution Change the constructor of the corresponding class to a private operation. Declare a static attribute `theInstance`, which refers to the single instance of the class. Initialize the pointer in the class declaration with null. Define a public static operation `getInstance()`, which returns the value of the attribute. When the operation is called for the very first time, the single instance of the class is constructed using the operator `new`. Furthermore, this instance is assigned to the attribute `theInstance`.

Idiom for implementing “Singleton” in C++ II

Example

```
class Singleton {
    static Singleton *theInstance;
    Singleton();
    public:
        static Singleton *getInstance() {
            if (! theInstance)
                theInstance = new Singleton;
            return theInstance;
        }
};
//...
Singleton* Singleton::theInstance = 0;
```

Idiom for Implementing "Singleton" in Smalltalk I

Name Singleton (Smalltalk)

Problem An implementation of the Singleton design pattern is needed to ensure that only one instance of a class exists at runtime.

Solution Override the operator `new` of the corresponding class such that it triggers an exception. Add the class attribute `TheInstance` to the class, which contains the single instance of the class. Implement the operation `getInstance()`, which returns this instance. When the operation is called for the very first time, the single instance of the class is constructed using the operator `super new`. Furthermore, this instance is assigned to the attribute `TheInstance`.

Idiom for Implementing “Singleton” in Smalltalk II

Example

```
new
   self error: 'cannot create new object'

getInstance
   TheInstance isNil ifTrue:
      [TheInstance := super new].
   ^TheInstance
```

Example of an Idiom in C++: Counted Pointer I

Example Problem of the C++ memory management. Several clients have a reference to a commonly used object. This issue leads to two unwanted situations:

1. A client object deletes the commonly used object while it is referenced by another client.
2. No client object references the commonly used object, but the object was not deleted.

Context Memory management of dynamically allocated, multiple-referenced instances of a class.

Example of an Idiom in C++: Counted Pointer II

Problem Objects will be passed as parameters to functions using pointers. The following *forces* rule:

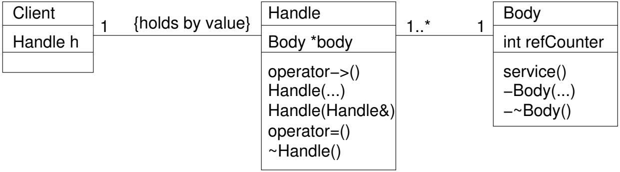
- several clients refer to the same object
- "dangling references" should be avoided
- object that are not referenced should be deleted
- solution should contain only a small portion of additional client code

Example of an Idiom in C++: Counted Pointer III

Solution

- counting of references of multiple-referenced objects
- *body class* will be extended by reference counter
- only a *handle class* is allowed to refer to objects of the body class
- objects will be passed as value parameters and hence automatically allocated and deleted
- handle class manages reference counter of body class instances
- by overloading the operator "`->`" in `object->operation()` using `operator->()` in the handle class, its instances can be used as if they were pointers on body class instances

Example of an Idiom in C++: Counted Pointer IV



Implementation

1. Declare the constructors and the destructor of the body class as private or protected methods to prevent uncontrolled creation and deletion of objects.
2. Declare the handle class as a friend class of the body class; hence it can access the features of the body class.
3. Extend the body class by a reference counter (`refCounter`).

Example of an Idiom in C++: Counted Pointer V

4. Add an attribute to the handle class pointing at a body object.
5. Implement the copy constructor (`Handle(Handle&)`) and the assignment operator of the handle class by copying the pointer to the body object and incrementing the reference counter. Implement the destructor (`~Handle`) of the handle class by decrementing the reference counter and deleting the body class object (if the reference counter reaches 0).
6. Implement the public arrow operator of the handle class as follows:

   ```
   Body* operator->() const { return body; }
   ```

7. Extend the handle class by one or more constructors, which create a body class instance the handle object points at. Each of these constructors initializes the reference counter of its body class object with 1.

Example of an Idiom in C++: Counted Pointer VI

Sample solution C++-Code ...

Variants *CountedBody*-Idiom (cf. Coplien 1992): each client has the illusion that it uses its own body class object, even though it is referenced by other clients. The body class object must be copied if a client modifies it.

What have we learned?

- Idioms are patterns on a low level of abstraction.
- They are tailor-made for specific (object-oriented) programming languages.
- They constitute concrete guidelines to solve specific programming problems in a specific programming language.

Summary

Patterns for different software development phases

- Architectural styles
  Structuring the software using components and connectors
- Design patterns
  Fine-grained design of architectural components, communication between components or objects
- Idioms
  Realization of a problem solution using a specific programming language

Conclusions

- There are patterns for practically all phases of software development.
- Patterns enable developers to construct software systematically.
- Patterns have the potential to improve not only the software development process, but also the resulting software products.