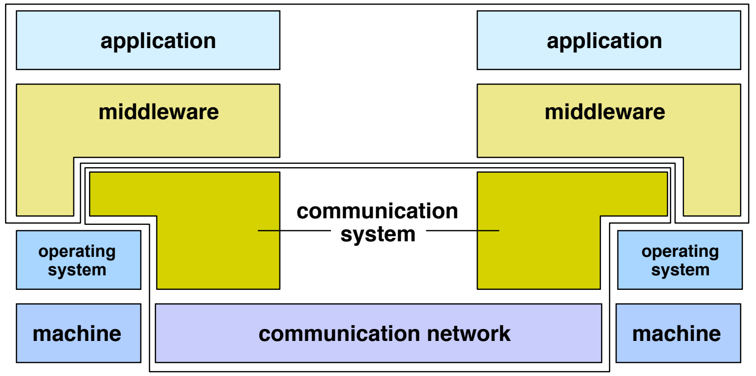
Towards Middleware Adaptation: Design Patterns

Vania Marangozova-Martin

acs.forge.imag.fr

Middleware: The General Picture



Middleware Goals (Principal Functions)

- Middleware has four main functions
  - High-level interface or API (Application Programming Interface) to applications
  - Mask heterogeneity of underlying hardware and software systems
  - Transparency of distribution
  - General/reusable services for distributed applications

Possible Classifications of Middleware…

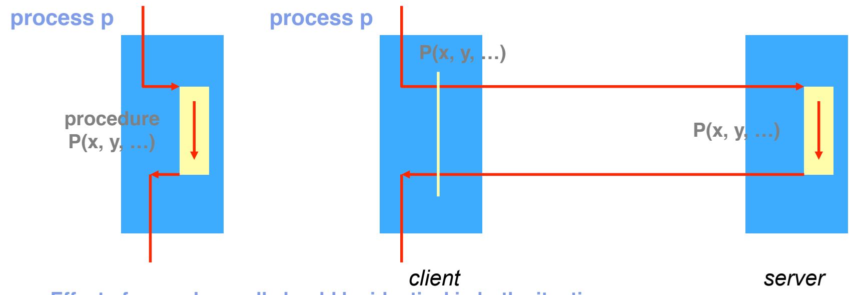
- Nature of communicating entities
  - Objects (e.g Java, C++)
  - Components (e.g J2EE, CORBA)
  - Processes (e.g MPI)
- Access mode to services
  - Synchronous (client-server)
  - Asynchronous (event-based)
  - Hybrid
- Other
  - Static vs. mobile entities
  - Guaranteed vs. non-guaranteed QoS

NO RIGOROUS CLASSIFICATION, DIFFERENT IMPLEMENTATIONS… HOW DO WE ADAPT?

Adapt on the basis of well-known/proven architectural/ implementation principles -> design patterns

A simple middleware example: RPC

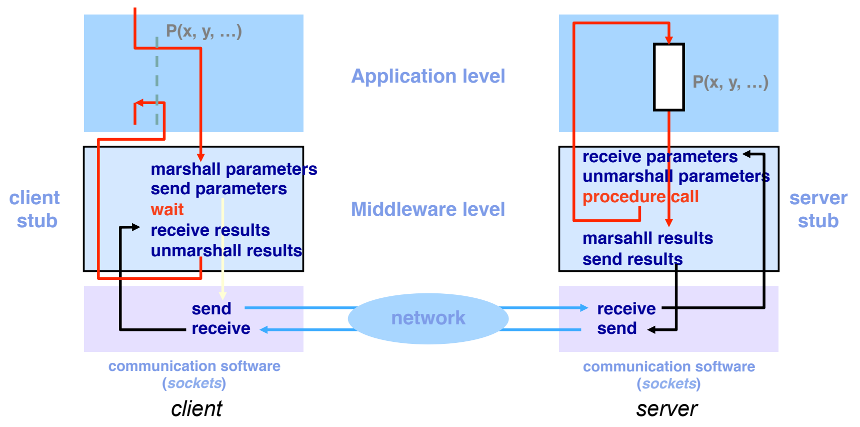
- Remote procedure call (RPC), a tool to build client-server distributed applications



Effect of procedure call should be identical in both situations.
Impossible in case of failures.

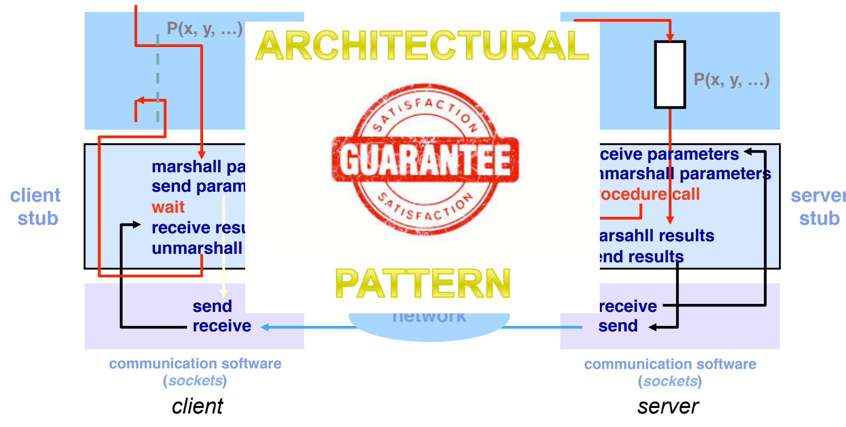
A simple middleware example: RPC (2)

- Implementation of remote procedure call



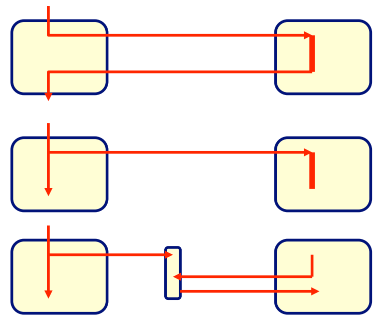
A simple middleware example: RPC (2)

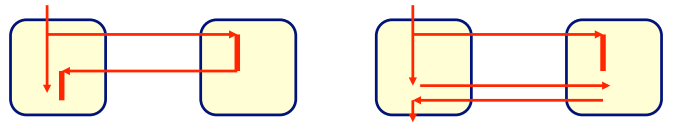
- Implementation of remote procedure call



Interaction patterns

- Synchronous
- Asynchronous
- Semi-synchronous





Tight coupling

RMI, CORBA, COM, …

Loose coupling

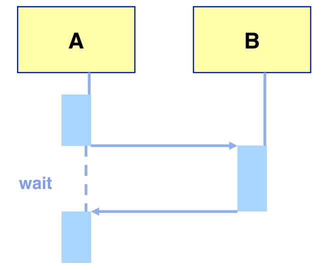
Events

Message queues

Combining synchronous - asynchronous

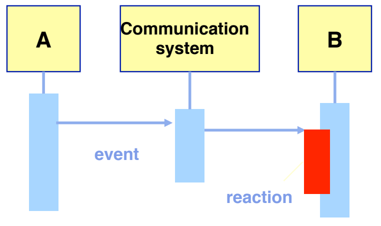
Interaction patterns (2)

- Synchronous interaction
  - Sender (client) blocks until it receives the results
  - Tight coupling

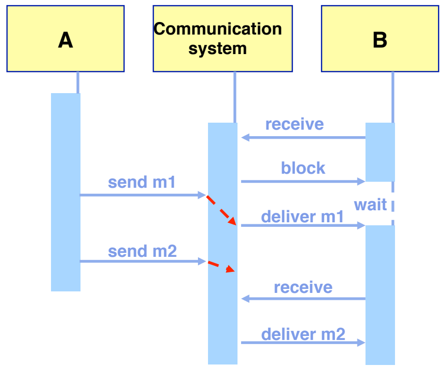


Interaction patterns (3)

- Asynchronous interaction
  - Parallel execution of sender (client) and receiver (server)
  - Loose coupling

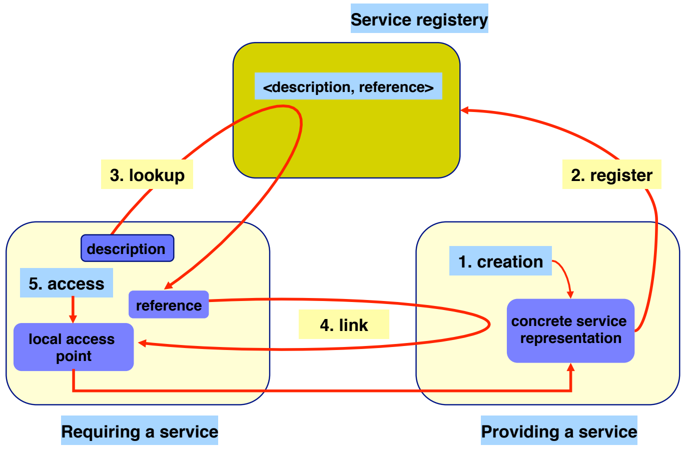


- Event-reaction



- Asynchronous messages

Access to a service – Example



Design patterns

- Definition [not only for software design]
  - Set of rules to provide a response to a family of needs that are specific to a given environment
  - Rules can have the form of
    - element definitions,
    - composition principles,
    - usage rules

Design patterns (2)

- Properties
  - A pattern is designed based on experience when solving a family of problems
  - A pattern captures common elements of solution
  - A pattern defines design principles, not implementations
  - A pattern provides help to documentation (e.g. terminology definition, formal description, etc.)

E. Gamma et. al. *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
F. Buschmann et. al. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996
D. Schmidt et. al. *Pattern-Oriented Software Architecture* - vol. 2, Wiley, 2000

Design patterns (3)

- Definition of a pattern
  - Context:
    - Situation rising a design issue
    - Must be as generic as possible (but not too generic)
  - Problem:
    - Specifications
    - Desired solution properties
    - Constraints on the environment
  - Solution:
    - Static aspects: components, relations between components (described with class or collaboration diagrams)
    - Dynamic aspects: behavior at runtime, life cycle (described with sequence or state diagrams)

F. Buschmann et. al. *Pattern-Oriented Software Architecture* - vol. 1, Wiley 1996

Examples of patterns

- *Proxy*
  - Pattern representative for remote access
- *Factory*
  - Pattern for managing object creation
- *Wrapper* [*Adapter*]
  - Pattern for interface transformation
- *Interceptor*
  - Pattern for service adaptation

These patterns are largely used in middleware implementations

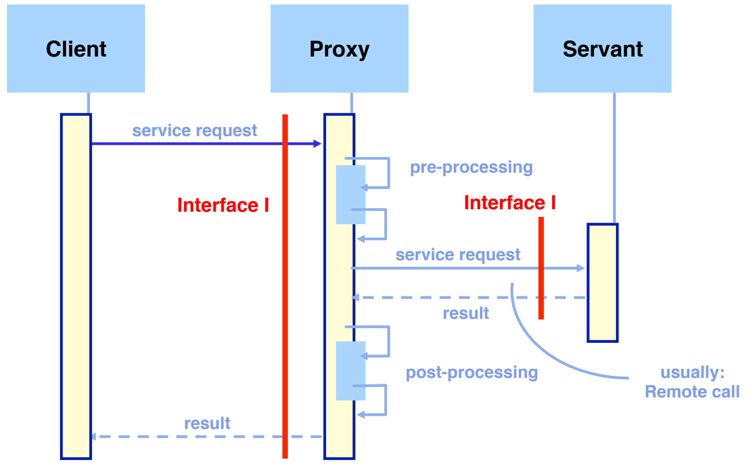
The *Proxy* Pattern

- Context
  - Applications as sets of distributed objects;
  - Client accesses services provided by a possibly remote object (servant)
- Problem
  - Define service access mechanisms that prevent
    - hand-coding server location in client code
    - having a detailed knowledge of communication protocols
  - Desired properties
    - efficient and dependable acces
    - simple programming model for client (ideally, no difference between local and remote service access)
  - Constraints
    - Distributed environment (no shared memory)

The *Proxy* Pattern (2)

- Solutions
  - Servant representative used locally at client-side (hide servant, and communication system to client)
  - Servant representative exposes same interface as servant
  - Define a uniform servant structure to ease its automatic generation

Use of *Proxy*



Question

- Does Java use proxies?

Examples of patterns

- *Proxy*
  - Pattern representative for remote access
- ***Factory***
  - **Pattern for managing object creation**
- *Wrapper* [*Adapter*]
  - Pattern for interface transformation
- *Interceptor*
  - Pattern for service adaptation

These patterns are largely used in middleware implementations

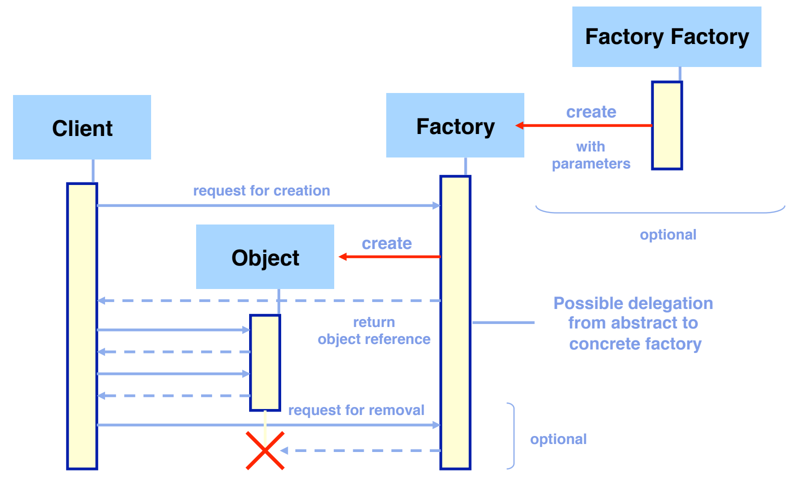
Factory

- Context
  - Application = set of objects in a distributed environment
- Problem
  - Dynamic creation of multiple instances of a class of objects
  - Desired properties
    - Instances may be parameterized
    - Easy evolution (no hand-coded decision)
    - Platform independance
  - Constraints
    - Distributed environment (no shared memory)

Factory (2)

- Solutions
  - *Abstract Factory*
    - Define an interface and a generic organization for object creation
    - Effective object creation is delegated to a concrete factory that implements creation methods

Use of *Factory*



Factory: A more concrete example

- Source: http://butunclebob.com ArticleS.UncleBob.AbstractFactoryDanielT

```
class Engine:

   def init(self, param):
      self.param = param

   def elsewhere(self):
      if self.param = "A":
         self.preParser = PreParserA()
      elif self.param = "B":
         self.preParser = PreParserB()

      # do some work

      if self.param = "A":
         self.parser = ParserA()
      elif self.param = "B":
         self.parser = ParserB()

      # do more work
```

- Engine that needs to support two languages, A and B

Factory: A more concrete example (2)

```
class Engine:
   def __init__(self, param):
       self.param = param

   def elsewhere(self):
       if self.param = "A":
          self.preParser = PreParserA()
       elif self.param = "B":
          self.preParser = PreParserB()

       # do some work

      if self.param = "A":
          self.parser = ParserA()
      elif self.param = "B":
          self.parser = ParserB()

      # do more work
```

```
        if self.param = "A":
                self.preParser = PreParserA()
                self.parser = ParserA()
        elif self.param = "B":
                self.preParser = PreParserB()
                self.parser = ParserB()
```

Combine the « ifs »?

May be too slow to process.

Factory: A more concrete example (3)

```
class Engine:
   def __init__(self, param):
       self.param = param

   def elsewhere(self):
       if self.param = "A":
          self.preParser = PreParserA()
       elif self.param = "B":
          self.preParser = PreParserB()

       # do some work

      if self.param = "A":
          self.parser = ParserA()
      elif self.param = "B":
          self.parser = ParserB()

      # do more work
```

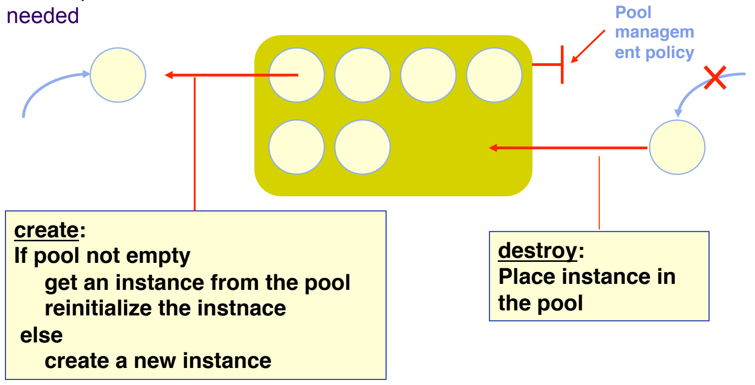
How will we manage with a third, 4th, 5th language???

```
class ParserFactoryA:
    def createPreParser(self): return PreParserA()
    def createParser(self): return ParserA()

class ParserFactoryB:
    def createPreParser(self): return PreParserB()
    def createParser(self): return ParserB()
```

```
class Engine:
    def __init__(self, param):
        if param = "A":
           self.parserFactory = ParserFactoryA()
        elif param = "B":
           self.parserFactory = ParserFactoryB()

    def elsewhere(self):
        self.preParser =
           self.parserFactory.createPreParser()

        # do some work

        self.parser =
           self.parserFactory.createParser()

        # do more work
```

Use of a Pool in a *Factory*

- Problem: online resource (e.g. objet) creation is expensive
- Objective: reduce costs underlying resource creation
- Technique: create a set of resources in advance and reuse them whenever needed



create:
If pool not empty
get an instance from the pool
reinitialize the instnace
else
create a new instance

destroy:
Place instance in the pool

Examples of use of *Pool*

- Memory management
  - *Pool* of memory regions (of possibly different sizes)
  - Prevent the overhead of garbage-collection
- Activity management
  - *Pool* of *threads*
  - Prevent overhead of online thread creation
- Communication management
  - *Pool* of connections
  - Prevent cost of online communication channel creation

BREAK

- 5 min

Examples of patterns

- *Proxy*
  - Pattern representative for remote access
- *Factory*
  - Pattern for managing object creation
- *Wrapper* [*Adapter, Decorator*]
  - Pattern for interface transformation
- *Interceptor*
  - Pattern for service adaptation

These patterns are largely used in middleware implementations

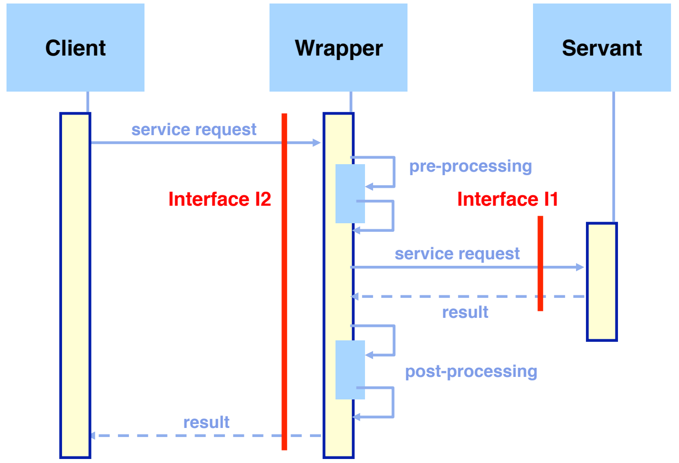
Wrapper (or *Adapter*)

- Context
  - Clients require services
  - Servants provide services
  - Services defined through interfaces
- Problem
  - Reuse an existing servant, while modifying its interface/ functions to satisfy client needs (or a subset of clients)
  - Desired properties: efficiency, reusable and adaptable to different needs

Wrapper (or *Adapter*) (2)

- Solutions
  - *Wrapper* isolates servant by intercepting calls to servant interface
  - Each call to servant interface is preceded by a prologue and followed by an epilogue in the *Wrapper*
  - Parameters of servant interface calls and results of calls can be modified

Use of *Wrapper*

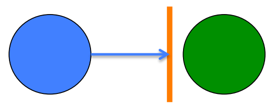


Wrapper Example (1)

```
public interface TransformText {
  String render(String aInputText);
}
```

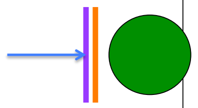
```
public static final class Echo implements TransformText{
    public String render(String aText) {
      return aText;
    }
  }
```

```
...
TransformText t = new Echo();
show(t.render("blah."));

//blah
```



Wrapper Example (2)

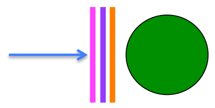
```
public static class BaseWrapper implements TransformText {
   BaseWrapper(TransformText aTransformText){
      fShowText = aTransformText;
   }

   public final String render(String aText) {
      String text = before(aText);
      text = fShowText.render(text); //call-forward
      return after(text);
   }

   /** This default implementation does nothing.*/
   String before(String aText){
      return aText;
   }

   /** This default implementation does nothing.*/
   String after(String aText){
     return aText;
   }
}
```



Wrapper Example (3)

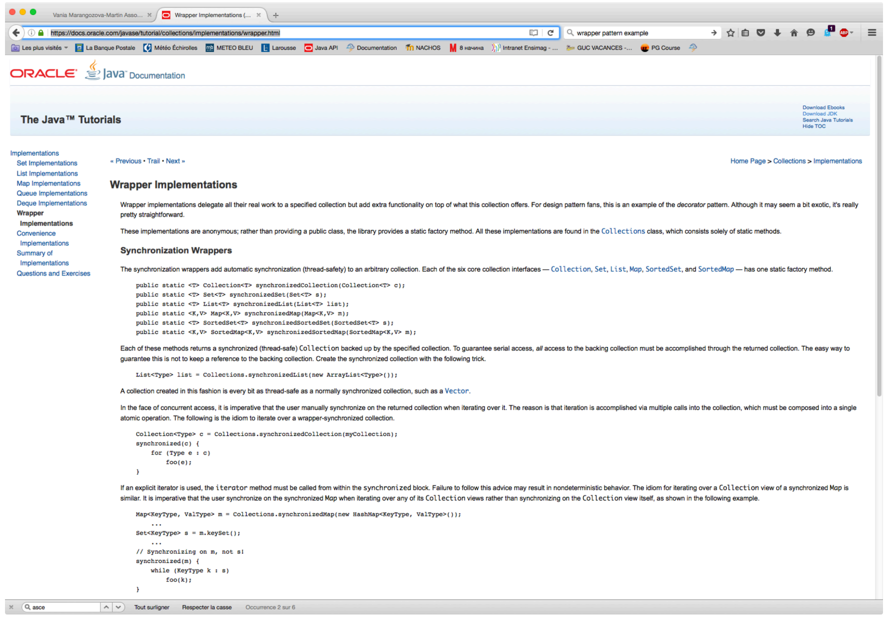
```
public static final class Capitalize extends BaseWrapper {
    Capitalize(TransformText aTransformText){
      super(aTransformText);
    }
    @Override String before(String aText) {
      String result = aText;
       if (aText != null){
         result = result.toUpperCase();
       }
       return result;
    }
  }
```

```
...
TransformText t = new Capitalize(Echo());
show(t.render("blah."));

//BLAH
```



A Real-World Example

- https://docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html



The Java™ Tutorials

Implementations
Set Implementations
List Implementations
Map Implementations
Queue Implementations
Deque Implementations
Wrapper Implementations
Convenience Implementations
Summary of Implementations
Questions and Exercises

« Previous • Trail • Next »

Home Page > Collections > Implementations

Wrapper Implementations

Wrapper implementations delegate all their real work to a specified collection but add extra functionality on top of what this collection offers. For design pattern fans, this is an example of the *decorator* pattern. Although it may seem a bit exotic, it's really pretty straightforward.

These implementations are anonymous; rather than providing a public class, the library provides a static factory method. All these implementations are found in the `Collections` class, which consists solely of static methods.

Synchronization Wrappers

The synchronization wrappers add automatic synchronization (thread-safety) to an arbitrary collection. Each of the six core collection interfaces — `Collection`, `Set`, `List`, `Map`, `SortedSet`, and `SortedMap` — has one static factory method.

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

Each of these methods returns a synchronized (thread-safe) `Collection` backed up by the specified collection. To guarantee serial access, *all* access to the backing collection must be accomplished through the returned collection. The easy way to guarantee this is not to keep a reference to the backing collection. Create the synchronized collection with the following trick.

```
List<Type> list = Collections.synchronizedList(new ArrayList<Type>());
```

A collection created in this fashion is every bit as thread-safe as a normally synchronized collection, such as a `Vector`.

In the face of concurrent access, it is imperative that the user manually synchronize on the returned collection when iterating over it. The reason is that iteration is accomplished via multiple calls into the collection, which must be composed into a single atomic operation. The following is the idiom to iterate over a wrapper-synchronized collection.

```
Collection<Type> c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
    for (Type e : c)
        foo(e);
}
```

If an explicit iterator is used, the `iterator` method must be called from within the `synchronized` block. Failure to follow this advice may result in nondeterministic behavior. The idiom for iterating over a `Collection` view of a synchronized `Map` is similar. It is imperative that the user synchronize on the synchronized `Map` when iterating over any of its `Collection` views rather than synchronizing on the `Collection` view itself, as shown in the following example.

```
Map<KeyType, ValType> m = Collections.synchronizedMap(new HashMap<KeyType, ValType>());
    ...
Set<KeyType> s = m.keySet();
    ...
// Synchronizing on m, not s!
synchronized(m) {
    while (KeyType k : s)
        foo(k);
}
```

Examples of patterns

- *Proxy*
  - Pattern representative for remote access
- *Factory*
  - Pattern for managing object creation
- *Wrapper [Adapter]*
  - Pattern for interface transformation
- *Interceptor*
  - Pattern for service adaptation

These patterns are largely used in middleware implementations

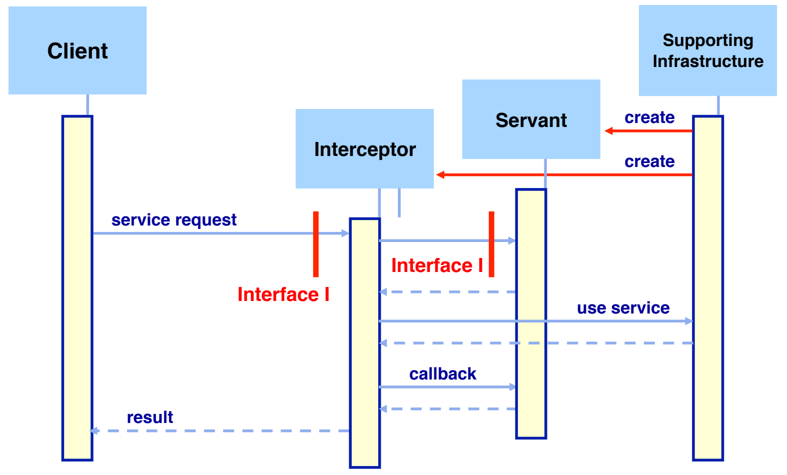
Interceptor

- Context
  - Provide services
    - Client-server, peer-to-peer, hierarchical
    - Uni- or bi-directional, synchronous or asynchronous
- Problem
  - Transform a service (add new functions)
    - Add a new processing level (cf. *Wrapper*)
    - Modify the target of the call
  - Constraints
    - Client and server programs must not be modified
    - Services may be dynamically added or removed

Interceptor (2)

- Solutions
  - Create interposition objects (statically or dynamically)
  - Interposition objets intercept service calls (and/or returns) and insert specific processing
  - Interposition objects may forward calls to other targets

Use of *Interceptor*



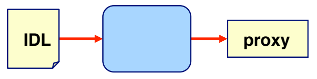
Comparison of patterns

- *Wrapper* vs. *Proxy*
  - *Wrapper* and *Proxy* have a similar structure
    - *Proxy* preserves interface ; *Wrapper* transforms interface
    - *Proxy* used for remote access; *Wrapper* used for local access
- *Wrapper* vs. *Interceptor*
  - *Wrapper* and *Interceptor* have a similar function
    - *Wrapper* transforms interface
    - *Interceptor* transforms function
- *Proxy* vs. *Interceptor*
  - *Proxy* is a simple form of *Interceptor*
    - An *Interceptor* may be added to a *Proxy* (*smart proxy*)

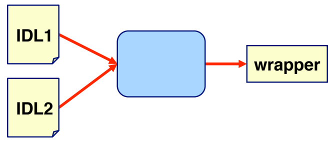
Implementation of patterns

- Automatic generation
  - From a declarative description

Proxy :



Wrapper :



Implementation of patterns (2)

- Optimizations
  - Eliminate indirections (performance overhead)
    - Shorten indirection chains
    - Code injection (insertion of generated code in application code)
    - Low-level code generation (e.g. Java bytecode)
    - Reversible techniques (for adaptation)

Patterns and adaptation

- Patterns may be known/detected =>
  - this knowledge may be used for adaptation
  - ! Optimization takes away the explicit pattern structures
    - More efficient
    - Less adaptable
- Patterns may explicitly target adaptation

Software frameworks

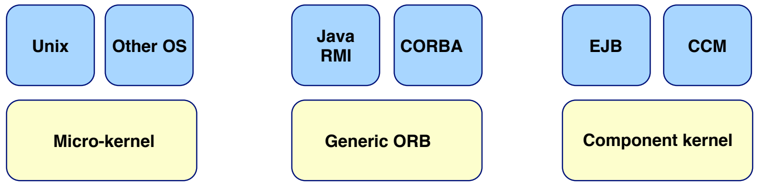
- Definition
  - A framework is a programme "squeleton" that can be used (adapted) for a famillity of applications
    - A framework usually comes with a set of defined components
      - Proven useful and reusable in multiple applications (services)
    - With rules of how to implement the application to respect/use the framework
  - A framework implements a model (not always explicit)
  - In object-oriented languages, a framework ususally consists in
    - A set of (abstract) classes that must be adapted (via inheritance) to different contexts
    - A set of usage rules for these classes
    - An SDK (Software Development Kit)

Software frameworks (2)

- Patterns and frameworks
  - Two techniques for reuse
  - Patterns reuse design principles
  - Frameworks reuse code implementation
  - A framework usually implements one or more patterns

Frameworks and personalities

- Motivation: reuse of generic mechanisms
  - A general framework implements entities defined in an abstract model
    - Criteria: genericity, modularity, adaptability
  - "Personnalities" use APIs of the general framework to build concrete implementations of the model
    - Advantages: reusability, reconfiguration
    - Issue: efficiency
- Exemples



References

- *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*, Jack Greenfield, Keith Short, 2004
- *Design Patterns for Dummies*, Steve Holzer
- *Design Patterns: Elements of Reusable Object-Oriented Software,* Erich Gamma, Richard Helm, Ralph Johnson, John Vissides
- http://www.informit.com/articles/article.aspx?p=1404056