21. Transparency Problems and the Decorator-Connector Pattern

A Design Pattern appearing in all classical component systems

Lecturer: Dr. Sebastian Götz

Prof. Dr. Uwe Aßmann
Technische Universität Dresden
Institut für Software- und Multimediatechnik
http://st.inf.tu-dresden.de/teaching/cbse
24. April 2017

1. Transparency Problems
2. Decorator-Connector Pattern
3. Interface Definition Languages
4. Location Transparency
5. Name Transparency and Trading
6. Example YP Service
7. Generic Skeletons
When the Object Management Group (OMG) was formed in 1989, interoperability was its founders primary, and almost their sole, objective:

- A vision of software components working smoothly together, without regard to details of any component's location, platform, operating system, programming language, or network hardware and software.

  - Jon Siegel
The Ladder of Composition Systems

Software Composition Systems
- Composition Language
- Invasive Composition
  - Piccola
  - Gloo

Aspect Systems
- Aspect Separation
- Crosscut graphs
- Aspect/J
  - AOM

View Systems
- Composition Operators
- Composition Filters
  - Hyperspaces

Architecture Systems
- Architecture as Aspect Connectors
- Darwin
- COSY
- BPMN
- ACME

Classical Component Systems
- Standard Components
- Reflection
- .NET CORBA
- Beans
- EJB

Object-Oriented Systems
- Objects as Run-Time Components
- C++
- Java

Modular Systems
- Modules as Compile-Time Components
- Shell scripts
- Modula
- Ada-85
21.1. Transparency Problems for COTS
Transparency Problems (Middleware Concerns)

- **A transparency problem describes software concerns that should be transparent (invisible, hidden) when you write or deploy a component.**
  - To solve a transparency problem, the component model requires different secrets

- **Content secrets**
  - **Language transparency**: interoperability of components using different programming languages
  - **Persistency transparency**
    - Hide whether server has persistent memory
  - **Lifetime transparency**
    - Hide whether server has to be started

- **Connection secrets**
  - **Location transparency**: distribution of programs
    - Hiding, where a program runs
  - **Naming transparency**: naming of services
    - Hiding, how a service is called
  - **Transactional transparency**
    - Hide whether server is embedded in parallel writes
Idea: Encapsulate Transparency Problems

**Components** encapsulate content secrets
- **Ports** abstract required and provided interface points of components (event channels, methods)
- Portsexplain the data-flow into and out of a component

**Connectors** are special communication components encapsulating connection secrets
- Connectors are attached to ports
- Connectors abstract from the concrete communication carrier
- Can be binary or n-ary
- Connector end is called a **role**
  - A role fits only to certain types of ports (typing)
21.2 The Decorator-Connector Pattern

- Connectors can hide implementation issues for connection transparency problems
Language Transparency With the Connector Pattern

The Connector Pattern (Double-Decorator Pattern, \(n\)-Decorator Pattern) can be used in a standard object-oriented language to implement connectors for classes and objects.

- **Stub**: Decorator of the client
  - Takes calls of clients in language \(A\) and sends them to the skeleton
- **Skeleton**: Decorator of the server
  - Takes those calls and sends the component implementation in language \(B\)

Language adaptation in Stub or Skeleton (or both)

- Adaptation deals with calling concepts, etc. (see above)
- Based on a mapping of language constructs from both languages, defined by an Interface Definition Language (IDL)
Basic Idea: Stubs and (Static) Skeletons as Decorators

- A typical instance of the Decorator pattern: two proxies on client and server
- Stub decorates skeleton, skeleton decorates server
The Decorator-Connector Pattern

- Client and server are connected via a layer of stubs and skeletons (the *connector*).
- The connector consists of two decorators of the server.
- Decorator chain is inherited.

```
$server.service(d : Data)$

<<client>>

Client

service(d : Data)

server.service(d);

<<server>>

Server

service(d : Data)

$next.service(d);$  

ServerDecorator

service(d : Data)

$next.service(d);$  

ServiceStub

service(d : Data)

next.service(d);

ServiceSkeleton

service(d : Data)

$next.service(d);$  

next
```
Connector consists of a Decorator chain, in a layer.

```java
// Client
<<client>>
c : Customer
startWork()

server : ServiceStub
createAccount()

// Server
<<server>>
bank : Bank
createAccount()

server.createAccount();
next

next.createAccount();

// Next

skeleton : ServiceSkeleton
createAccount()

next.createAccount();
next.next.createAccount();
```
Layered Decorators (Object Diagram)

- More decorators can be stuffed into the connector in additional layers:

```java
<<client>>
c : Customer

startWork()

server.createAccount();

<<server>>
bank : Bank

createAccount()

server

stub1 : ServiceStub1

createAccount()

next

next

next

next

server.createAccount();

next.createAccount();

next.createAccount();

next.createAccount();

stub2 : ServiceStub2

createAccount()

next

next

skeleton1 : ServiceSkeleton1

createAccount()

next

next

server

skeleton2 : ServiceSkeleton2

createAccount()

next

next

next

next

next.createAccount();

next.createAccount();

next.createAccount();

next.createAccount();
```

Component-Based Software Engineering (CBSE)
Decorator vs Proxy vs Adapters vs Chain

Why should it be a Decorator?
- Decorators allow for stacking of connectors (layering)
- Proxy pattern: just one representative, no stacking possible
  - However, from the client and server's perspective, stub and skeletons are Proxies
- Adapter: Adapted interface must be different from Adaptee
- Chain: In a Chain, the processing may stop (not here..)

However, Connectors can use all other basic “representer” patterns
- Adapter-Connector: adapts required interface to server additionally
- Chain-Connector: may stop processing
- Proxy-Connector: just one layer possible
Tasks of the Layers

In a component model, every layer of decorator-pairs is devoted to a specific task for transparency (middleware concern)

- Language mappings (language interoperability)
- Distribution handling (serialization, deserialization)
- Names (name mapping, name search)
- Persistence
- Transactions
- etc.

Layers can be composed (stacked) freely
Containers – Infrastructure for all Connectors

- A **container** of a server component is an infrastructure for all connectors at run-time (all decorators/proxies).
  - Creation (server component factories for service families)
  - Transactions (begin, rollback, commit)
  - Persistence (activate, passivate)
- The container is an instance of the Façade design pattern (DPF)
Who Realizes Stubs and Skeletons?

- **Programmer**
  - Much handcrafting, using Decorator pattern. Boring and error prone

- **Generator:**
  - **Stub**
    - Export interface is component dependent, independent of source language
    - Implementation is source language dependent
  - **Skeleton**
    - Import interface is component dependent, independent of source language
    - Implementation is target language dependent

- **Idea:** Generate export and import interfaces of Stub and Skeleton out of a component interface definition
  - Take generic language adapter for the implementation
21.3 Interface Definition Languages for Mapping Different Languages

- Language mediation with the „star approach“
Transparency Problem 1: Language Transparency

- **Calling concept**
  - Procedure, Co-routine, Messages, ...

- **Calling conventions**
  - Call by name, call by value, call by reference, ...

- **Calling implementation**
  - Parameters on the stack, in registers, allocation and de-allocation

- **Data types**
  - Value and reference objects
  - Arrays, union, enumerations, classes, (variant) records, ...
  - Kind of inheritance (co-variance, contra-variance, ...)

- **Data representation**
  - Coding, size, little or big endian, ...
  - Layout of composite data

- **Runtime environment**
  - Memory management, garbage collection, lifetime ...
Language Mediation - Options In General

- Direct language mapping (full graph of language relationships):
  - 1:1 adaptation of pairs of languages: $O(n^2)$
- Mapping to common language ("star approach"):  
  - Adaptation to a general exchange format: $O(n)$  
  - CORBA IDL
- Compiling to common basic type system
  - .NET, WSDL
Language Mediation – Common Basic Type System

- Compiling to common basic type system:
  - Standardize to a single format (like in .NET): $O(1)$ but very restrictive, because the languages become very similar
Language Mediation – WSDL

- Web Service Definition Language (WSDL) uses a similar concept as .NET, but encodes everything as XML.
Solutions in Classical Component Systems

- **Calling concept:**
  - Standardized by the communication library (RPC)

- **Calling conventions:**
  - Standardized by the communication library (EJB - Java, DCOM - C)
  - Implementation for every single language (Corba)

- **Calling implementation:**
  - Standardized by the communication library (EJB - Java, DCOM - C)
  - Implementation for every single language (Corba)

- **Data types:**
  - Standard (EJB – Java types)
  - Adaptation to a general exchange format (interface definition language, IDL)
    - CORBA IDL
    - Web Service Definition Language (WSDL)

- **Data representation:**
  - Standard (EJB – Java representation, DCOM – binary standard)
  - Adaptation to a general format (IDL 2 Language mapping)

- **Runtime environment**
  - Standard by services of the component systems
Type Mapping with the CORBA IDL

- An IDL language defines the
  - Interfaces of components
  - Data types of parameters and results

- Language independent type system
  - General enough to capture all data types in a programming language
  - IDL mediates between type systems of these languages

- Procedure of construction
  - Define component interface with IDL
  - Generate stubs and skeletons with required languages using an IDL compiler
  - Implement the frame (component) in respective language (if possible reusing some other, predefined components)
Ex.: Types in the CORBA Interface Definition Language

// IDL specification scheme
modules <identifier> {  
    <type declarations>
    <constant declarations>
    <exception declarations>

// classes
interface <identifier> : <inheritig-from> {  
    <type declarations>
    <constant declarations>
    <exception declarations>
// methods
    optype <identifier>(<parameters>) {  
        ...
    }
    ....
}

module HelloWorld {
    interface SimpleHelloWorld {
        string sayHello();
    }
};
Generation of Stubs and Skeletons from CORBA IDL

- Generation is done for every involved host programming language (HPL)
- Interface Repository is queried for component interfaces (introspection)
Stubs and Skeletons for Language Mediation

**Stub**
- Map data to an exchange format (IDL)
- Call Skeleton

**Skeleton**
- Receive call from stub
- Retrieve data from the exchange format (IDL), transform it into language 2
21.4 Location Transparency
Transparency Problem 2: Distribution

- **Location transparency**: interoperability of programs independently of their execution location

- **Problems to solve**
  - Transparent basic communication
    - Transparently initiate a local/remote call
    - Transparently transport data locally or remotely via a network
    - Transparent references
  - Distributed systems are heterogeneous
    - Platform transparent, concurrent execution?
    - So far we handled platform transparent design of components
  - Usual aspects in distributed systems
    - Transactions
    - Synchronization
    - ...
Transparent Local/Remote Calls

- Communication over proxies/decorators
  - Proxies redirect call locally or remotely on demand
  - Proxies always local to the caller

- RPC for remote calls to a handler
  - Handler always local to the callee

- We reuse Stubs and Skeletons
A variant of the Connector pattern, using remote procedure call (RPC) between the decorators.
Stubs and Skeletons for Distribution

Component-Based Software Engineering (CBSE)

Site 1

Client

Stub

Skeleton

Server

Site 2

Language 1

Map data / call to a byte stream (marshalling, serializing)
Exchange format
Send RPC

Language 2

Receive RPC
Retrieving data / call from the byte stream (unmarshalling, deserializing)
Exchange format

RPC
or with separate serializers/deserializers

**Site 1**
- **Client**
- **Stub**
- **Serializer**

**Language 1**
- Map data / call to a *byte stream* (marshalling, serializing)
- Exchange format
- Send RPC

**Site 2**
- **Deserializer**
- **Skeleton**
- **Server**

**Language 2**
- Receive RPC
- Retrieving data / call from the *byte stream* (unmarshalling, deserializing)
- Exchange format

**RPC**
21.5 Name Transparency and Trading

• Mapping names to locations by name servers
Transparency Problem 3: The Reference Problem (Name Transparency)

- How to reference something?
  - Target of calls (services)
  - Call by reference parameters and results
  - Reference data in composite parameters and results

- Scope of references
  - Thread/process
  - Computer
  - Agreed between communication partners
  - Net wide

- How to handle references transparently?
Approach: Global Adresses

- World wide unique *logical* addresses
  - e.g., computer address + local address
  - URL (Uniform Resource Locators), URI (Uniform Resource Identifiers)
  - CORBA IORs (Interoperable Object References)
  - Global file names, e.g., with AFS (Andrew File system)
  - Names in a global cloud file system (DropBox, Skydrive, etc.)
  - Names in a private cloud file system http://sparkleshare.org/

- Mapping tables for local references
  - Logical to physical
  - Consistent change of local references possible

- One server decorator per computer manages references
  - 1:n relation decorator to skeletons
  - 1:m relation skeletons to component objects
  - Lifecycle and garbage collection management
  - Identification (Who is this guy ...)
  - Authorization (Is he allowed to do this ...)
Name Service

- Name to Location
- Located in the container as an associative array (map)
Distributed name service (name to location):

- If name of server is known, search for the right site providing a desired component.
**Extended name service, dynamic call:**
- If name of server is **not** known, search for the right service with faceted feature description
Traders as Generalized Name Servers

- **Trader service, traded call** map symbolic service descriptions (service properties) to name or location
  - Search for a server component with known properties, but *unknown* name
  - Server components register at a *trader* with name, reference, and lookup properties (metadata)
    - The trader has a component repository (*registry*)
    - Instead of names, lookup of service matches properties (metadata)
    - Return reference (site and service)
  - Matching relies on standardized properties
    - Terminology, Ontology in facets (see “Finding components”)
    - Functional properties (domain specific functions ...)
    - Non-functional properties (quality of service ...)

![Diagram](image)
Remark: Skeletons, NameServers, and Containers

- Can be started and consulted by skeletons
- May offer many other aid functionality
  - Transactions: consistent management of multiple clients and service requests
  - Security
  - Persistence
  - Interception (hooks into which new functionality can be entered)
  - Support for aspects
What Classical Component Systems Provide

- Technical support: remote, language and platform transparency
  - Stub, Skeleton
    - One per component (technique: IDL compiler)
    - Generic (technique: reflection and dynamic invocation)
  - Decorators on client and server site
    - Individual
    - Generic (technique: Name services)

- Economically support: reusable services
  - name, trader, persistency, transaction, synchronization
Summary

Component systems provide many component secrets
- Location, language and platform transparency
- Transactional, persistence, security, name service

Component secrets are realized with the Connector Pattern (Stub, Skeleton-Pattern)
- One pair or tuple of Decorators per component in a layer, but several layers, stacking Decorators on top of each others
- On the server side, adapters help to make services generic
- Decorators, Proxies, Adapters, Chains on client and server site

Generated by IDL compiler
- Is the IDL compiler essential?
- No! Generic stubs and skeletons are possible, too. Technique: Reflective invocation
A More Simple Connector with Server Interface (Alt. 2, with Abstract Interface)

- Client and server are connected via a layer of stubs and skeletons (the connector)
- Server, Stubs and Skeletons inherit from same interface (not a Decorator!) → this cannot be layered
Example: A Remote Yellow Page Service

• with remote access, serialization
Remote Yellow Page Service

- Basic design without Serialization/Deserialization
Remote Yellow Page Service

- With Serialization/Deserialization

```
ClientStub.lookup("MyName");
ClientStub.store("MyName", "name");
ClientStub.lookup("MyName");
```
interface IYellowPageService {
    String SERVICE_NAME = "Yellow Pages";
    String lookup(String name);
    void store(String name, String value);
}
class YellowPageService implements IYellowPageService {
    private Hashtable<String, String> cache = new Hashtable<String, String>();
    private DataBase db = ...;
    public String lookup(String name) {
        String res = cache.get(name);
        if (res == null)
            res = db.lookup(name);
        if (res != null) {
            cache.put(name, res);
        }
        return res;
    }

    public void store(String name, String value) {
        cache.put(name, value);
        db.store(name, value);
    }
}
Client

- Wants to transparently use the Yellow Page service

Site 1
- Client
- Stub
- Serializer
- Deserializer

Site 2
- Skeleton
- Server
Example Client

Client calls stub with service interface:

```java
class Client {
    ...
    // returns client stub
    IYellowPageService yps =
        YellowPageFactory.create();
    ...
    String res = yps.lookup("MyName");
    ...
}

class YellowPageFactory {
    public IYellowPageService create() {
        return new YellowPageStub();
    }
}
```
Stub (client side)

- Realizes 1:1 mapping of client to service component
- Uses 1:1 mapping of clients to stubs
Example Client Stub - Implementation

class YellowPageStub implements IYellowPageService {
    private Integer logicalAddress = new Integer(-1);

    public YellowPageStub() {
        logicalAddress = (Object) ClientSerializer.invoke(
            IYellowPageService.SERVICE_NAME, logicalAddress, "new", null);
    }

    public String lookup(String name) {
        Object res = ClientSerializer.invoke(IYellowPageService.SERVICE_NAME,
            logicalAddress, "lookup", new Object[]{name});
        return (String)res;
    }

    public void store(String name, String value) {
        ClientSerializer.invoke(IYellowPageService.SERVICE_NAME,
            logicalAddress, "store", new Object[]{name, value});
    }
}
Scenario with Second Stub (client site)

- By using the Decorator pattern, stubs can be stacked onto each other
- Every stub solves another transparency problem (middleware concern)
// new stub: encryption decorator
class YellowPageStubEncryption implements IYellowPageService {
    private IYellowPageService clientDec;

    // Security: encryption, decryption
    private String encrypt(String name);
    private String decrypt(String name);

    // client-side constructor
    public YellowPageStubEncryption() {
        clientDec = new YellowPageStub();
    }

    // lookup function, with encryption, decryption
    public String lookup(String name) {
        String res = clientDec.lookup(encrypt(name));
        return decrypt(res);
    }

    // store
    // ...
}
Client-side Serializer

- Manages the basic communication on client side
- Is called from the client stubs
- Can be hidden in a Decorator (1:1), but can be also shared by all stubs
Example Client Serializer

class ClientSerializer {
    public static Object invoke(String service, Integer address, String method, Object[] args) {

        Socket s = new Socket("yp-st.inf.tu-dresden.de", 1234);
        ObjectOutputStream os = new ObjectOutputStream(s.getOutputStream());
        ObjectInputStream is = new ObjectInputStream(s.getInputStream());
        os.writeObject(service);
        os.writeObject(address);
        os.writeObject(method);
        if (args != null) {
            os.writeObject(args);
        }
        os.flush();
        Object result = is.readObject();
        s.close();
        return result;
    }
}
Server-side Deserializer

- Manages the basic communication on server side
- Calls the service skeletons (1:n mapping)
Deserializer listens on the network is shared between different services

- interprets incoming service names
- can create/invoke several service skeletons
- lives always, but hides lifetime of the server

```java
class ServiceDeserializer {
    public void run() {
        ServerSocket server = new ServerSocket(1234);
        Socket client = server.accept();
        ObjectInputStream is = new ObjectInputStream(client.getInputStream());
        ObjectOutputStream os = new ObjectOutputStream(client.getOutputStream());
        while (true) {
            String service = (String) is.readObject();
            if (service.equals(IYellowPageService.SERVICE_NAME)) {
                handleYellowPage(os, is);
            } else if (service.equals(IPhoneBook.SERVICE_NAME)) {
                handlePhoneBook(os, is);
            } else {
                System.err.println("Unknown service.");
            }
        }
    }
}
```
private void handleYellowPage(ObjectOutputStream os, ObjectInputStream is) {
    Integer address = (Integer) is.readObject();
    if (address == -1) { // creation of the service
        YellowPageSkeleton skeleton = new YellowPageSkeleton();
        os.writeObject(skeleton.getLogicalAddress());
    } else { // service query: interpretation of the symbolic service name
        IYellowPageService yp = new YellowPageSkeleton(address);
        String method = (String) is.readObject();
        Object[] args = (Object[]) is.readObject();
        if (method.equals("lookup")) {
            String res = yp.lookup((String)args[0]); // finally: call the service
            os.writeObject(res);
        } else if (method.equals("store")) {
            yp.store((String)args[0], (String)args[1]);
            os.writeObject(null);
        } else
            System.err.println("Unknown service method.");
    }
    os.flush();
}
Skeleton (Server side)

- Manages service components of server on server side
- 1:1 mapping to service component
public class YellowPageSkeleton implements IYellowPageService {
    private static Hashtable<Integer, IYellowPageService> yellowPageServices =
        new Hashtable<Integer, IYellowPageService>();
    private Integer logicalAddress;

    public YellowPageSkeleton() {
        this(new Integer(yellowPageServices.size()));
        yellowPageServices.put(logicalAddress, new YellowPageService());
    }

    public YellowPageSkeleton(Integer address) {
        logicalAddress = address;
    }

    public Integer getLogicalAddress() { return logicalAddress; }

    public String lookup(String name) {
        IYellowPageService service = yellowPageServices.get(logicalAddress);
        return service.lookup(name);
    }

    public void store(String name, String value) {
        IYellowPageService service = yellowPageServices.get(logicalAddress);
        service.store(name, value);
    }
}
Creation of YP Service

Client

Stub

Decorator Client Site

Decorator Server Site

Skeleton

Server Impl

create

invoke ("create")

Socket Communication Call object

new

new

return Service CompImpl

return Stub Object

res handle

Socket Communication handle
Call (Lookup) YP Service

Component-Based Software Engineering (CBSE)
Generic Skeletons

• Mapping names to locations by name servers
Reflection & Reflective Invocation

Reflection
- to inspect the interface of an unknown component
- for automatic/dynamic configuration of server sites
- to call the inspected components

Access to interfaces with IDL
- Standardize an IDL run time representation and access
- Define a IDL specification for IDL representation and access
- Store IDL specifications in *interface repositories* which can be introspected
A **generic skeleton** is a special case of a name service: using reflection to look up the name for a method

```java
class ReflectiveSkeleton {

    // serverObjects is the server implementation repository
    static ExtendendHashtable serverObjects = new ExtendedHashtable();
    ObjectOutputStream os;
    ObjectInputStream is;
    ...

    public Object handleGeneric() { ...
        Integer addr = (Integer) is.readObject(); // handler
        String mn = (String) is.readObject(); // method name
        Class[] pt = (Class[]) is.readObject(); // parameter types
        Object[] args = (Object[]) is.readObject(); // parameters

        // get server object reference by reflective call to implementation repository
        Object o = serverObjects.getComponent(addr);
        Method m = o.getClass().getMethod(mn, pt); // method object by reflection
        Object res = m.invoke(o, args); // method call by reflection
        os.writeObject(res);
        os.flush();
    }

    ...
```
Appendix
The Decorator Design Pattern

➢ (Repetition from DPF in winter)
Decorator Pattern

A Decorator is a *skin* of another object

It is a 1-ObjectRecursion (i.e., a restricted Composite):
- A subclass of a class that contains an object of the class as child
- However, only one composite (i.e., a delegatee)

Combines inheritance with aggregation
- Inheritance from an abstract Handler class
- That defines a contract for the mimiced class and the mimicing class

```
:Client
  ref
  A:Decorator
    hidden
  B:Decorator
    hidden
  C:RealObject
```
Decorator – Structure Diagram

Component-Based Software Engineering (CBSE)

MimickedClass

mimickedOperation()

ConcreteMimickedClass

mimickedOperation()

Decorator

mimickedOperation()

ConcreteDecoratorA

mimickedOperation()

ConcreteDecoratorB

mimickedOperation()

super.mimickedOperation();
additionalStuff();
Example: Decorator for Widgets

```
Widget
   draw()

TextWidget
   draw()

WidgetDecorator
   draw()

Frame
   draw()

Scrollbar
   draw()

super.draw();
   drawFrame();

mimiced.draw();
```
Decorator for Persistent Objects

Component-Based Software Engineering (CBSE)

`Record`
- `access()`

`TransientRecord`
- `access()`

`PersistentDecorator`
- `access()`

`PersistentRecord`
- `access()`

`PersistentRead OnlyRecord`
- `access()`
- `boolean loaded()`
- `load()`

`PersistentRecord`
- `access()`
- `boolean loaded()`
- `boolean modified()`
- `load()`
- `dump()`

```java
if (!loaded()) load();
super.access();
if (modified()) dump();
```
Purpose Decorator

- For extensible objects (i.e., decorating objects)
  - Extension of new features at runtime
  - Removal possible
- Instead of putting the extension into the inheritance hierarchy
  - If that would become too complex
  - If that is not possible since it is hidden in a library
Variants of Decorators

- If only one extension is planned, the abstract super class Decorator can be omitted; a concrete decorator is sufficient.
- Decorator family: If several decorators decorate a hierarchy, they can follow a common style and can be exchanged together.
- Decorators can be chained to each other.
- Dynamically, arbitrarily many new features can be added.
The End

- Many slides courtesy to Prof. Welf Löwe, Växjö University, Sweden.