Patterns for Bridging Architectural Mismatch

Prof. Dr. U. Aßmann
Chair for Software Engineering
Faculty of Informatics
Dresden University of Technology
D. Garlan, R. Allen, J. Ockerbloom. Architectural mismatch – or why it is so hard to build systems out of existing parts. Int. Conf. On Software Engineering (ICSE 95) http://citeseer.nj.nec.com/garland95architectural.html

GOF – Adapter.
Further Literature
Goal

► Understand architectural mismatch
► Understand design patterns that bridge architectural mismatch
Architectural Mismatch

- Case study of Garlan, Allen, Ockerbloom 1995
- Building the architectural system Aesop
Architectural Mismatch

- Aesop was built out of 4 off-the-shelf components
  - OBST: an object-oriented C++ database
  - Interviews and Uniframe, a windowing toolkit
  - Softbench, an event bus (event-based mediator)
  - RPC interface generator of Mach (MIG)

- All subsystems written in C++ or C

- First Aesop version took 5 person years, and was still sluggish, very large

- Problems can be characterized in terms of components and connections
Classification of Different Assumptions of the COTS

- Different Assumptions about the *component model*
  - Infrastructure
  - Control model
  - Data model
- Different assumptions about the *connectors*
  - Protocols
  - Data models
- Different assumptions about the *global architectural structure*
- Different assumptions about the *construction process*
Different Assumptions about the Component Model

- A component model assembles information and constraints about the nature of components
  - Nature of interfaces
  - Substitutability of components

- Here:
  - Infrastructure
  - Control model
  - Data model

- More in “Component-Based Software Engineering”, summer semester
Different Assumptions about the Component Infrastructure

- Components assume that they should provide a certain infrastructure, which the application does not need
  - OBST provides many library functions for application classes; Aesop needed only a fraction of those
- Components assume they have a certain infrastructure, but it is not available
  - Softbench assumed that all other components have access to an X window server (for communication)
Assumptions on Control Model

- COTS think differently in which components have the main control
  - Softbench, Interviews, and MIG have an ever-running event loop inside
  - They call applications with callbacks (observer pattern)
- However, they use different event loops:
  - Softbench uses X window event loop
  - MIG and Interviews have their own ones
  - The event loops had to be reengineered, to fit to each other
Assumptions on Data Model

► Different assumptions about the data
  ▪ Uniframe: hierarchical data model
  ▪ Manipulations only on a parent, never on a child
  ▪ However, the application needed that
  ▪ Decision: rebuild the data model from scratch, is cheaper than modification
Assumptions about the Connectors
Protocols

- Softbench works asynchronously; which superimposes concurrency to tools
  - Softbench is a mediator between tools
- 2 kinds of interaction protocols
  - Request/Reply (callback, observer): tool requests a service, registers a callback routine, is called back by Softbench
  - Notify via Softbench
Protocols

- Softbench works asynchronously; which superimposes concurrency to tools, when messages of different tools are crossing.
Data Formats

- Components also have different assumptions what comes over a channel (a connection).
  - Softbench: Strings
  - MIG: C data
  - OBST: C++ data

- Requires translation components
  - When accessing OBST, data must be translated all the time
  - This became a performance bottleneck
Assumptions about the Global Architecture

► OBST

- Assumes a database-centered architecture
- Assumes independence of client tools
- And provides a transaction protocol per single tool, not per combination of tools
- Doesn't help when tools have interactions
Assumptions about the Building Process

- Assumptions about the library infrastructure
- Assumptions about a generic language (C++)
- Assumptions about a tool specific language
- Combination is fatal:
  - Some component A may have other expectations on the generated code of another component B as B itself
  - Then, the developer has to patch the generated code of A with patch scripts (another translation component)
Proposed Solutions of [Garlan]

- Make architectural assumptions explicit
  - Problem: how to document or specify them?
  - Many of the aforementioned problems are not formalized
  - Implicit assumptions are a violation of the information hiding principle, and hamper variability
- Make components more independent of each other
- Provide bridging technology
  - For building language translation components (compiler construction, compiler generators, XML technology)
- Distinguish architectural styles (architectural patterns) explicitly
  - Distinguish connectors explicitly
- Solution: design patterns serve all of these purposes
Usability of Extensibility Patterns

- All extensibility patterns can be used to treat architectural mismatch
- Decorator for wrapping and architectural mismatch
- ChainOfResponsibility as filter for objects
- Proxy for translation between data formats
- Observer for additional extensions, listening to the events of the subject
- Visitor for extension of a data structure hierarchy with new algorithms
- Bridge for factoring designs (making components independent)
Adapter
Object Adapter

- An object adapter is a proxy
  - That maps one interface to another.
  - Or a protocol
  - Or a data format

- An adapter cannot easily map control flow to each other
  - Since it is passed *once* when entering the adapted class
Object Adapter

Client → Goal

Goal

operation()

.Adapter

operation()

adaptedObject

AdaptedClass

specificOperation()

adaptedObject.specificOperation()

Adapted class does not inherit from goal.

Decorator-like inheritance

Object adapters use delegation
Example: Use of an External Class Library For Texts

```
GraficObject
    frame()
    createManipulator()

DrawingEditor
    *
    GraficObject

Text
    frame()
    createManipulator()

Linie
    frame()
    createManipulator()

TextDisplay
    largeness()

return new TextManipulator

return text.largeness()
```

External Library
Adapters for COTS

- Adapters are often used to adapt components-off-the-shelf (COTS) to applications.
- For instance, an EJB-adapter allows for reuse of an Enterprise Java Bean in an application.
BillingApplication \* \rightarrow Bill

addItem(Item)
calculateSum()

OtherBill
addItem(Item)
calculateSum()

EJBBill
fetchBean()
addItem(Item)
calculateSum()

.. contact EJBHome for EJB...
.. if not there, create EJBOBJECT

.. EJBOBJECT = fetchBean();
.. addItem(EJBOBJECT, Item)

.. EJBOBJECT = fetchBean();
.. sum up (EJBOBJECT)
A Remark to Adapters in Component Systems

- Component models define *standard, unspecific* interfaces
  - E.g., EJBHome / EJBOBJECT
- Classes usually define *application-specific* interfaces
- To increase reuse of classes, the Adapter pattern(s) can be used to map the application-specific class interfaces to the unspecific component interfaces
- Example:
  - In the UNIX shell, all components obey to the pipe-filter interfaces *stdin, stdout, stderr* (untyped channels or streams of bytes)
  - The functional parts of the components have to be *mapped* by some adapter to the unspecific component interfaces.
Adapters and Decorators

- Similar to a decorator, an adapter inherits its interface from the goal class
- Hence, adapters can be *inserted* into inheritance hierarchies later on
Facade

- A facade is an object adapter that hides a complete set of objects (subsystem)
  - Or: a proxy that hides a subsystem
  - The facade has to map its own interface to the interfaces of the hidden objects
Facade Hides a Subsystem

**Abstract**

Facade

```
operation()
```

**Client**

```
HiddenClass1
specificOperation()
```

```
HiddenClass2
specificOperation()
```

```
HiddenClass3
specificOperation()
```

**Concrete Facade**

```
operation()
```

```
adaptedObject.specificOperation()
```

```
adaptedObject2.specificOperation()
```

```
adaptedObject3.specificOperation()
```

**HiddenSubsystem**

```
adapted Object1
```

```
adapted Object2
```

```
adapted Object3
```

```
....
adaptedObject.specificOperation()
adaptedObject2.specificOperation()
....
```
Instead of delegation, class adapters use multiple inheritance
More than one goal class may exist.
Every goal class plays a role of the concrete object (see later).
2-Way Adapter for Coupling of Class Hierarchies

SuperClass A

SubClass A

GoalClass A

operation()

SuperClass B

GoalClass B

operation2()

SubClass B

Adapter

operation()

operation2()
2-Way Decorator and Adapter for Coupling of Class Hierarchies

SuperClass A

SubClassA

GoalClass A

operation()

GoalClassB

operation2()

SuperClass B

SubClass B

2WayAdapterDecorator

operation()

operation2()
2-Way Decorator and Adapter for Coupling of Class Hierarchies

DataGenerator

TestDataGenerator

GeneratorAdapter

operation()

StrategyAdapter

operation2()

GenerationStrategy

Exhaustive

**GeneratorStrategy**

operation()

operation2()

GeneratorStrategy can be used to have several strategies in a chain of decorators
Mediator (Broker)
Mediator (Broker)

- A mediator is an n-way proxy for communication
  - Combined with a Bridge
- A mediator serves for
  - *Anonymous* communication
  - *Dynamic* communication nets
Mediator

ConcreteMediator

ConcreteColleague1

ConcreteColleague2

AColleague

AConcreteMediator

AColleague

AColleague

AColleague

AColleague

AColleague

Typical Object Structure:
Mediator As n-Proxy and Bridge
Proxy object hides all communication partners
- Every partner uses the mediator object as proxy
- Clear: real partner is hidden

Bridge links both communication partners
- Both mediator and partner hierarchies can be varied
Web Service Brokers

**ConcreteServiceMediator**
- `buy()`
- `query(Widget)`

**WebServiceMediator**
- `buy()`
- `query(WebService)`

**WebService**
- `query()`

**Google**
- `search()`

**HotelBooking**
- `search()`
- `reserve()`
- `buy()`

mediator

买的

查询（WebService）
Summary

► Architectural mismatch between components and tools consists of different assumptions about components, connections, architecture, and building procedure
► Design patterns, such as extensibility patterns or communication patterns, can bridge architectural mismatches
► Reuse of COTS is much better
The End