Softwaretechnik / Software-Engineering
Lecture 13: Architecture and Design Patterns

Prof. Dr. Andreas Podelski,
Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Introduction and Vocabulary

• Software Modelling
  • model; views / viewpoints; 4+1 view
  • Modelling structure
    • (simplified) class & object diagrams
    • (simplified) object constraint logic (OCL)

Principles of Design

• Principles of (Good) Design
  • modularity, separation of concerns
  • information hiding and data encapsulation
  • abstract data types, object orientation

Design Patterns

• Architecture Patterns
  • Layered Architectures, Pipe-Filter, Model-View-Controller.
• Design Patterns
  • Strategy, Examples
• Libraries and Frameworks

Goals and Relevance of Design

• The structure of something is the set of relations between its parts.
• Something not built from (recognisable) parts is called unstructured.

Design...
(i) structures a system into manageable units (yields software architecture),
(ii) determines the approach for realising the required software,
(iii) provides hierarchical structuring into a manageable number of units at each hierarchy level.

Oversimplified process model “Design”:

 req.  design  arch.  designer
 modulespec.  impl.  code  programmer
Components can be verified/validated in isolation.

- IOW: other components cannot depend on details they are not supposed to. Without other components noticing, unintentionally changed hidden solutions may be goals/advantages.

- Information hiding is about which data or operations other components may use of this component. Making explicit is about (e.g., how data is stored and accessed, how operations are implemented).

- Note that information hiding need not know: what is hidden is information which other components need to know about the module that is not in the module’s interface specification. As little as possible about the module’s inner workings, and other modules are prevented from using information hiding: Users do not write to bank accounts directly, only bank clerks do. Real-world example: Users do not write to bank accounts directly, only bank clerks do. Similar direction: Do not access data (variables, files, etc.) directly where needed, but encapsulate the data in a component which offers operations to access (read, write, etc.) the data.

- By now, we only discussed the need to know principle. The “need to know principle” / information hiding is called the “need to know principle.” One should also consider the "need to know principle". IEEE 610.12 (1990) and Parnas (1972) in SW engineering. (examples later).

4.) Data Encapsulation

- As long as the interface does not change, it should be possible to test old and new versions of a module together. Many programming languages and systems offer means to use these means. As an example, consider the following module interface: 

```
mod = {  
  name = name,  
  data = data,  
  operations = operations,  
}  
```

- In this example, the module name and the set of operations are available to the user. The data, however, are not. The user cannot access the data directly. Instead, the user can only manipulate the data through the operations provided by the module. This is an example of data encapsulation.

- The benefit of data encapsulation is that it helps to separate the implementation of a module from its interface. This means that changes to the module’s implementation do not necessarily affect its interface, and thus its users. This can make it easier to update and maintain the module, as well as to reuse it in different contexts.

- Data encapsulation is a fundamental principle in software engineering and is often discussed in the context of object-oriented programming. It is one of the key concepts in the design and implementation of software systems.
Example Module: List of Names

A Possible Implementation: Plain List of Duplicates

Change Function: Support Duplicate Name

Information hiding and data encapsulation:

Similar direction:

Evolution:

Information hiding and data encapsulation:

Examples

Good modules.

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:

Information hiding and data encapsulation:

Examples

Evolution:
Data Encapsulation / Information Hiding

Object-oriented by example

Information hiding and data encapsulation by modules, negative effects when requirements change.
Introduction

Over decades of software engineering, many clever, proved and tested designs of solutions for particular problems emerged.

Question: can we generalise, document and re-use these designs?

Goals:
• "don’t re-invent the wheel",
• benefit from "clever", from "proven and tested", and from "solution".

Architectural Pattern: An architectural pattern expresses a fundamental structural organisation schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Buschmann et al. (1996)

Using an architectural pattern implies certain characteristics or properties of the software (construction, extendibility, communication, dependencies, etc.), determines structures on a high level of the architecture, thus is typically a central and fundamental design decision.

The information that (where, how, ...) a well-known architecture / design pattern is used in a given software can make comprehension and maintenance significantly easier, avoid errors.

Example: Layered Architectures

- Züllighoven, 2005:
  A layer whose components only interact with components of their direct neighbour layers is called protocol-based layer. A protocol-based layer hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

- Example: The ISO/OSI reference model.

  7. Application
  6. Presentation
  5. Session
  4. Transport
  3. Network
  2. Data link
  1. Physical

- Object-oriented layer: interacts with layers directly (and possibly further) above and below.

  Rules: the components of a layer may use
  - only components of the protocol-based layer directly beneath, or
  - all components of layers further beneath.

Examples: GNOME etc. Applications

  GNOME
  
  GTK+
  
  GDK
  
  ATK
  
  Cairo
  
  GLib
  
  GIO
  
  Pango

Example: Three-Tier Architecture

- Desktop Host
  
  Presentation tier
  
  Application Server
  
  (business) logic tier
  
  Data tier
  
  Database Server
  
  DBMS

- Ludewig and Lichter, 2013

  Presentation layer (or tier):
  user interface; presents information obtained from the logic layer to the user, controls interaction with the user, i.e. requests actions at the logic layer according to user inputs.

  Logic layer:
  core system functionality; layer is designed without information about the presentation layer, may only read/write data according to data layer interface.

  Data layer:
  persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.

Examples: Web-shop, business software (enterprise resource planning), etc.
Layered Architectures: Discussion

7. Application
6. Presentation
5. Session
4. Transport
3. Network
2. Data link
1. Physical

GNOME etc.
Applications
GTK+
GDK ATK
Cairo GLib
GIO Pango

Desktop Host
presentation tier
Application Server
(business) logic tier
data tier
Database Server
DBMS

• Advantages:
• protocol-based: only neighbouring layers are coupled, i.e. components of these layers interact,
• coupling is low, data usually encapsulated,
• changes have local effect (only neighbouring layers affected),
• protocol-based: distributed implementation often easy.

• Disadvantages:
• performance (as usual) — nowadays often not a problem.

Example: Pipe-Filter

Compiler
lexical analysis
(llexer)
syntactical analysis
(parser)
semantical analysis
code generation
ASCII
Tokens
AST
dAST
Sourcecode
Objectcode
Errormessages

• Disadvantages:
• if the filters use a common data exchange format, all filters may need changes
• if the format is changed, or need to employ (costly) conversions.
• filters do not use global data, in particular not to handle error conditions.

Example: Model-View-Controller

controller
(view
uses
change of
visualisation
manipulation of
data
notification of
updates
access to data

https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg Der Geusa, CC-BY-SA-2.5
**Example: Model-View-Controller**

- **Advantages**:
  - One model can serve multiple view/controller pairs;
  - View/controller pairs can be added and removed at runtime;
  - Model visualisation always up-to-date in all views;
  - Distributed implementation (more or less) easily.

- **Disadvantages**:
  - If the view needs a lot of data, updating the view can be inefficient.

**Design Patterns**

- In a sense the same as architectural patterns, but on a lower scale.
- Often traced back to (Alexander et al., 1977; Alexander, 1979).

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. (Gamma et al., 1995)

**Example: Pattern Usage and Documentation**

![Diagram](https://commons.wikimedia.org/wiki/File:Maschinenleitstand_KWZ.jpg)

**Strategy**

**Problem**

The only difference between similar classes is that they solve the same problem by different algorithms.

**Solution**

- Have one class *StrategyContext* with all common operations.
- Another class *Strategy* provides signatures for all operations to be implemented differently.
- From Strategy, derive one sub-class *ConcreteStrategy* for each implementation alternative.
- *StrategyContext* uses concrete Strategy-objects to execute the different implementations via delegation.

**Structure**

```
StrategyContext
 + contextInterface()

Strategy
 + algorithm()

ConcreteStrategy1
 + algorithm()

ConcreteStrategy2
 + algorithm()
```
Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

**Problem**
- The behaviour of an object depends on its (internal) state.
- The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.

**Example**
- All GUI object displaying a file system need to change if files are added or removed.

**Problem**
- Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.
- Appearance and state of different means of interaction (menus, buttons, input fields) in a graphical user interface (GUI) should be consistent in each interaction state.

**Other Patterns: Singleton and Memento**

**Singleton**
- Of one class, exactly one instance should exist in the system.
- Example: Print spooler.

**Memento**
- The state of an object needs to be archived in a way that allows to reconstruct this state without violating the principle of data encapsulation.
- Example: Undo mechanism.

**Design Patterns: Discussion**

*"The development of design patterns is considered to be one of the most important innovations of software engineering in recent years."* (Ludewig and Lichter, 2013)

**• Advantages**
- (Re-)use the experience of others and employ well-proven solutions.
- Can improve on quality criteria like changeability or re-use.
- Provide a vocabulary for the design process, thus facilitates documentation of architectures and discussions about architecture.
- Can be combined in a flexible way, one class in a particular architecture can correspond to roles of multiple patterns.
- Helps teaching software design.

**• Disadvantages**
- Using a pattern is not a value as such. Having too much global data cannot be justified by "but it's the pattern Singleton".
- Again: reading is easy, writing need not be. Here: Understanding abstract descriptions of design patterns or their use in existing software may be easy — using design patterns appropriately in new designs requires (surprise, surprise) experience.
• (Class) Library: a collection of operations or classes offering generally usable functionality in a re-usable way.
  Examples:
  • libc — standard C library (is in particular abstraction layer for operating system functions),
  • libz — compress data.
  • libxml — read (and validate) XML file, provide DOM tree.

• Framework: class hierarchies which determine a generic solution for similar problems in a particular context.
  Example: Android Application Framework

• The difference lies in flow-of-control: library modules are called from user code, frameworks call user code.

• Product line: parameterised design/code ("all turn indicators are equal, turn indicators in premium cars are more equal").

Quality Criteria on Architectures

• testability
  • architecture design should keep testing (or formal verification) in mind (buzzword "design for verification"),
  • high locality of design units may make testing significantly easier (module testing),
  • particular testing interfaces may improve testability (e.g. allow injection of user input not only via GUI; or provide particular log output for tests).

• changeability, maintainability
  • most systems that are used need to be changed or maintained, in particular when requirements change,
  • risk assessment: parts of the system with high probability for changes should be designed such that changes are possible with acceptable effort (abstract, modularise).

• portability
  • porting: adaptation to different platform (OS, hardware, infrastructure).
  • systems with a long lifetime may need to be adapted to different platforms over time, infrastructure like databases may change

Note:
• a good design (model) is first of all supposed to support the solution, it need not be a good domain model.

References