Softwaretechnik / Software-Engineering

Lecture 11: Architecture & Design

2015-06-16

Prof. Dr. Andreas Podelski,
Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Top Topic Area Architecture & Design: Content

- Introduction and Vocabulary
- Principles of Design
  - (i) modularity
  - (ii) separation of concerns
  - (iii) information hiding and data encapsulation
  - (iv) abstract data types, object orientation
- Software Modelling
  - (i) views and viewpoints, the 4+1 view
  - (ii) model-driven/-based software engineering
  - (iii) Unified Modelling Language (UML)
  - (iv) modelling structure
    - (simplified) class diagrams
    - (simplified) object diagrams
    - (simplified) object constraint logic (OCL)
  - (v) modelling behaviour
    - (simplified) communicating finite automata
    - Uppaal query language
    - basic state-machines
    - an outlook on hierarchical state-machines
- Design Patterns

Introduction
An outlook on

- model-driven software engineering
- Architectural Description
- the 4+1 view

is described by

views & viewpoints

architectural description

(i) encapsulates a subset of the system's functionality and/or data, (ii) restricts access to that subset via an explicitly defined interface, and (iii) has explicitly defined dependencies on its required execution context.

The interface of a component provides the services of the component to the component's environment and/or requires services needed by the component from there.

The interface (of component) communicates with each other.

Architecture is the fundamental organization of a system embodied in its components, the externally visible properties of those elements, and the relationships among them.

The software architecture of a program or computing system is a model—document, product or other artifact—to communicate and record a system's architecture. An architectural description conveys a set of characteristics of a system or component. (2) The result of the process in (1).

A component is a software system, component, module, interface, or software and may be subdivided into other components.

A collection of components organized to accomplish a specific function or set of functions.

A set of software units and their relations, if they together serve a common purpose.

— The fundamental organization of a system embodied in its components, the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them.

A system into a set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.

— The boundary between two communicating components.

— A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.

— An architectural entity that communicates with other components.

— A model—document, product or other artifact—to communicate and record a system’s architecture. An architectural description conveys a set of characteristics of a system or component. (2) The result of the process in (1).

— The boundary across which two independent entities meet and interact or combine with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.

— A logically separable part of a program.

— A program unit that is discrete and identifiable with respect to compiling, assembling, linking, and loading.

— A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.

— A model—document, product or other artifact—to communicate and record a system’s architecture. An architectural description conveys a set of characteristics of a system or component. (2) The result of the process in (1).

— A program unit that is discrete and identifiable with respect to compiling, assembling, linking, and loading.

— A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.

— A model—document, product or other artifact—to communicate and record a system’s architecture. An architectural description conveys a set of characteristics of a system or component. (2) The result of the process in (1).

— A program unit that is discrete and identifiable with respect to compiling, assembling, linking, and loading.

— A set of operations and data visible from the outside only in so far as explicitly permitted by the programmers.
1.) Modularisation

- **Modular Decomposition**
  - The process of breaking a system into components to facilitate design and development; an element of modular programming.
  - IEEE 610.12 (1990)

- **Modularity**
  - The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
  - IEEE 610.12 (1990)

- So, modularity is a property of an architecture.

- **Goals of Modular Decomposition**
  - The structure of each module should be simple and easily comprehensible.
  - The implementation of a module should be exchangeable; information on the implementation of other modules should not be necessary. The other modules should not be affected by implementation exchanges.
  - Modules should be designed such that expected changes do not require modifications of the module interface.
  - Bigger changes should be the result of a set of minor changes. As long as the interface does not change, it should be possible to test old and new versions of a module together.

2.) Separation of Concerns

- **Separation of Concerns**
  - A fundamental principle in software engineering:
  - Each component should be responsible for a particular area of tasks.
  - Components which try to cover different task areas tend to be unnecessarily complex, thus hard to understand and maintain.

- **Criteria for Separation/Grouping**
  - In object-oriented design, data and operations on that data are grouped into classes.
  - Sometimes, functional aspects (features) like printing are realised as separate components.
  - Separate functional and technical components.
  - Example: logical flow of (logical) messages in a communication protocol (functional) vs. exchange of (physical) messages using a certain technology (technical).

- Assign flexible or variable functionality to own components. Example: different networking technology (wireless, etc.).

- Assign functionality which is expected to need extensions or changes later to own components.

- Separate system functionality and interaction. Example: most prominently graphical user interfaces (GUI), also file input/output.

3.) Information Hiding

- By now, we only discussed the grouping of data and operations. One should also consider accessibility.
- The "need to know principle" is called information hiding in SW engineering. (Parnas, 1972)

- **Information Hiding**
  - A software development technique in which each module’s interfaces reveal as little as possible about the module’s inner workings, and other modules are prevented from using information about the module that is not in the module’s interface specification.
  - IEEE 610.12 (1990)

- Note: what is hidden is information which other components need not know (e.g., how data is stored and accessed, how operations are implemented).
- In other words: information hiding is about making explicit for one component which data or operations other components may use of this component.

- **Advantages/Goals**
  - Hidden solutions may be changed without other components noticing, as long as the visible behaviour stays the same (e.g. the employed sorting algorithm).
  - IOW: other components cannot (unintentionally) depend on details they are not supposed to.
  - Components can be verified/validated in isolation.

4.) Data Encapsulation

- Similar direction: data encapsulation (examples later).

- 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.

- Real-World Example: Users do not write to bank accounts directly, only bank clerks do.
A Classification of Modules

• Information hiding and data encapsulation: by modules
  • Design and implementation of data modules
    • Numerical data type modules
    • String data type modules
    • Abstract data type modules
    • Functional modules
  • Data type modules
    • Encapsulate data
    • Do not access data (variables, files, etc.) directly where needed
    • Avoid dirty data
  • Functional modules
    • Do not have “memory” or state, that is, behaviour of offered functionality does not depend on prior use
    • Do not rely on order of access

• Functionality of code modules
  • Functions
    • Single responsibility
    • Make functions self-contained
  • Classes
    • Single responsibility
    • Make classes self-contained

• Functionality of state modules
  • Data items
    • Single responsibility
    • Make data items self-contained

• Functionality of control modules
  • Control flow
    • Single responsibility
    • Make control flow self-contained

• Functionality of design modules
  • Design
    • Single responsibility
    • Make design self-contained
# Abstract Data Type

## Data Encapsulation + Information Hiding

```cpp
#include <vector>
#include <string>

using namespace std;

class Names {
public:
    void insert(int it, int n) {
        // Implementation
    }

    void begin() {
        // Implementation
    }

    void end() {
        // Implementation
    }

    std::vector<std::string> names;

    int count;

    Names(int i) {
        count = i;
        // Implementation
    }

    void dump() {
        // Implementation
    }

    int std::string get(int i) {
        // Implementation
    }

    bool operator==(int i) {
        // Implementation
    }

    void new_Names() {
        // Implementation
    }

    void remove(int i) {
        // Implementation
    }

    void dump() {
        // Implementation
    }

private:
    int old_count;
};
```

---

## Change Implementation: Support Duplicates

- Made implementation more efficient by storing duplicate names.

## Change Interface: Support Duplicate Names

- Updated interface to support duplicate names.

---

## Werner: 1

- Werner: 1

## Berg: 1

- Berg: 1

## Schulz: 1

- Schulz: 1

## Meyer: 1

- Meyer: 1

## Abstract Data Type

- Abstract Data Type

---

## Data Encapsulation + Information Hiding

- Data Encapsulation + Information Hiding

---

## Change Implementation: Support Duplicates

- Changed implementation to support duplicates.

---

## Change Interface: Support Duplicate Names

- Updated interface to support duplicate names.
remove(2);
remove(1);
dump();
eprintf(0);

void Names::insert(std::string n) {
    #include<vector>
    #include<string>
    std::vector<std::string> names;

eprintf(new);

  
  WE Development Hiding and Data Encapsulation,
  Abstract Data Types

  V
eprintf(Tell Them What You've Told Them
example: Design-Models in Construction Engineering

1. Requirements

(i) the original that are relevant in the modelling context are represented,
(ii) the image or archetype the model is,

Three properties are constituent:

• information hiding
• data encapsulation
• separation of concerns

Cost shall be in budget.

Each room shall have a door.

Bathroom shall have a window.

Furniture shall fit into living room.

Shall fit on given piece of land.

Bathroom windows

1. Requirements

Cost shall be in budget.

Each room shall have a door.

Bathroom shall have a window.

Furniture shall fit into living room.

Shall fit on given piece of land.
Views and Viewpoints

A representation of a whole system from the perspective of a related set of concerns.

IEEE 1471 (2000)

A viewpoint — A specification of the conventions for constructing and using a view. A pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis.

IEEE 1471 (2000)

• A perspective is determined by concerns and information needs:
  • team leader, e.g., needs to know which team is working on what component,
  • operator, e.g., needs to know which component is running on which host,
  • developer, e.g., needs to know interfaces of other components.
  • etc.

An Early Proposal: The 4+1 View
(Kruchten, 1995)

Logical View
Development View
Process View
Physical View
Scenarios

end-user
functionality
programmers,
software management

system engineers,
.topology,
communication

integration
performance, scalability

• system view: how is the system under development integrated into (or seen by) its environment; with which other systems (including users) does it interact.

• static view: (≈ developer view): components of the architecture, their interfaces and relations. Possibly: assignment of development, test, etc. onto teams.

• dynamic view: (≈ process view): how and when are components instantiated and how do they work together at runtime.

• deployment view: (≈ physical view): how are component instances mapped onto infrastructure and hardware units.

"Purpose of architecture: support functionality; functionality is not part of the architecture. "

Process and Physical View


Example: modern cars
• large number of electronic control units (ECUs) spread all over the car,
• which part of the overall software is running on which ECU?
• which function is used when? Event triggered, time triggered, continuous, etc.?

For, e.g., a simple smartphone app, process and physical view may be trivial or determined by the employed framework — so no need for (extensive) particular documentation.

Structure vs. Behaviour

Definition.

Software is a finite description \( S \) of a (possibly infinite) set \( \{S\} \) of (finite or infinite) computation paths of the form \( \sigma_0 \alpha_1 \cdots \sigma_i \alpha_i \cdots \) where

• \( \sigma_i \in \Sigma \), \( i \in \mathbb{N}_0 \), is called state (or configuration), and

• \( \alpha_i \in \mathcal{A} \), \( i \in \mathbb{N}_0 \), is called action (or event).

The (possibly partial) function \( \{ \cdot \} : S \mapsto \{S\} \) is called interpretation of \( S \).

• Form of the states \( \Sigma \) (also actions \( \mathcal{A} \)):
  • structure of \( S \)

• Computation paths \( \pi \) of \( S \):
  • behaviour of \( S \)

(Harel, 1997) proposes to distinguish constructive and reflective descriptions of behaviour:

• constructive: "constructs [of description] contain information needed in executing the model or in translating it into executable code."
  → how things are computed.

• reflective (or assertive):
  "[description used] to derive and present views of the model, statically or during execution, or to set constraints on behavior in preparation for verification."
  → what should (or should not) be computed.

Note: No sharp boundaries! (would be too easy...)

Model-Driven Software Engineering

- Jacobson et al., 1992: "System development is model building."

Model-driven software engineering (MDSE): everything is a model.

Model-based software engineering (MBSE): some models are used.

Idea Structure

Declarative Behaviour

Constructive Behaviour

Structure

Implementation

Requirements refine requirements/model

generate/program

A Brief History of the Unified Modelling Language (UML)

- Boxes/lines and finite automata are used to visualise software for ages.
- Mid 1980's: Statecharts (Harel, 1987), StateMate™ (Harel et al., 1990)
- Early 1990's, advent of Object-Oriented -Analysis/Design/Programming — Inflation of notations and methods, most prominent:
  - Object-Modeling Technique (OMT) (Rumbaugh et al., 1990)
  - Booch Method and Notation (Booch, 1993)

http://wikimedia.org (CC nc-sa 3.0, User:AutumnSnow)
Modelling languages:

- Unified Modelling Language (UML)
- Communicating finite automata (v)
- Statecharts (1980s)
- StateMate (1987)

**Unified Modelling Language (UML):**

- 1.x: Joint effort of "the three amigos" (Jacobson et al., 1992; Rumbaugh et al., 1990; Booch, 1993)
- 2.x: Not-for-profit, international, open membership standards published by Object Management Group (OMG) starting 2005

**Object-Oriented Techniques:**

- Use case model realized by Booch Method and Notation (Booch, 1990)
- Class, object, analysis, design, implementation
- Archetypes or images (Dobing and Parsons, 1985)
- Testing, managed units, separation of concerns

**Software Engineering:**

- Views and viewpoints, the 4+1 view (Jacobson et al., 1992)
- Information hiding / data encapsulation (Jacobson et al., 1992)
- Mathematical, formal, abstract, pragmatic attribute reduction attribute

**References:**

- Dobing and Parsons (1985)
- Jacobson et al. (1992)
- Rumbaugh et al. (1990)
- Booch (1993)
- Harel et al. (1987)
- Harel et al. (1990)

**Model-driven/-based software engineering:**

- Software Modelling: Principles of Design
- Model-driven software development: views and viewpoints, e.g. 4+1
- Mathematical, formal, abstract, pragmatic attribute reduction attribute


