Softwaretechnik / Software-Engineering

Lecture 11: Architecture & Design

2015-06-16

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

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
    a) (simplified) class diagrams
    b) (simplified) object diagrams
    c) (simplified) object constraint logic (OCL)
  (v) modelling behaviour
    a) communicating finite automata
    b) Upaal query language
    c) basic state-machines
    d) an outlook on hierarchical state-machines
• Design Patterns
Survey: Previous Experience

Requirements Engineering

Programming

Design Modelling

Software Quality Assurance

Content

• Vocabulary
  • (software) system, component
  • module, interface
  • design, architecture

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

• Software Modelling
  • model
  • views & viewpoints
  • the 4+1 view
  • model-driven software engineering

• An outlook on UML
Introduction
Vocabulary

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

*IEEE 1471 (2000)*

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

This purpose is in general complex, it usually includes, next to providing one (or more) executable program(s), also the organisation, usage, maintenance, and further development.

*(Ludewig and Lichter, 2013)*

**component**—One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components.

*IEEE 610.12 (1990)*

**software component**—An architectural entity that

1. encapsulates a subset of the system’s functionality and/or data,
2. restricts access to that subset via an explicitly defined interface, and
3. has explicitly defined dependencies on its required execution context.

*(Taylor et al., 2010)*

**module**—(1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from an assembler, compiler, linkage editor, or executive routine.

(2) A logically separable part of a program.

*IEEE 610.12 (1990)*

**interface**—A boundary across which two independent entities meet and interact or communicate with each other.

*(Bachmann et al., 2002)*

**interface (of component)**—The boundary between two communicating components.

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 the requirement.

*(Ludewig and Lichter, 2013)*
**Even More Vocabulary**

**design**—
(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1).

IEEE 610.12 (1990)

**architecture**— The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.

IEEE 1471 (2000)

**software architecture**— The software architecture of a program or computing system is the structure or structures of the system which comprise software elements, the externally visible properties of those elements, and the relationships among them.

(Bass et al., 2003)

**architectural description**— A model--document, product or other artifact--to communicate and record a system’s architecture. An architectural description conveys a set of views each of which depicts the system by describing domain concerns.

(Ellis et al., 1996)

---

**Once Again, Please**

System
\[ \frac{\text{consists of 1 or more}}{\text{Component}} \]
Component
\[ \frac{\text{has}}{\text{Component Interface}} \]
Software System
\[ \frac{\text{Software Component}}{\text{may be a Module}} \]
Software Architecture

Architecture
\[ \frac{\text{is the result of}}{\text{Design}} \]
Architectural Description
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":

Content

- Vocabulary
  - (software) system, component
  - module, interface
  - design, architecture

- Principles of (Good) Design
  - modularity
  - separation of concerns
  - information hiding and data encapsulation
  - abstract data types, object orientation
  - information hiding / data encapsulation / etc. by example

- Software Modelling
  - model
  - views & viewpoints
  - the 4+1 view
  - model-driven software engineering

- An outlook on UML
1.) Modularisation

**modular decomposition** – The process of breaking a system into components to facilitate design and development; an element of modular programming.

**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.

- 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** is 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
  - assign flexible or variable functionality to own components.
  - assign functionality which is expected to need extensions or changes later to own components.

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.
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.

- **Information hiding** and **data encapsulation** – when enforced technically (examples later) – usually **come at the price** of worse efficiency.
- It is more efficient to read a component's data directly than calling an operation to provide the value: there is an overhead of one operation call.
- Knowing how a component works internally may enable more efficient operation.
  
  **Example**: if a sequence of data items is stored as a singly-linked list, accessing the data items in list-order may be more efficient than accessing them in reverse order by position.

  **Good modules** give usage hints in their documentation (e.g. C++ standard library).

  **Example**: if an implementation stores intermediate results at a certain place, it may be tempting to 'quickly' read that place when the intermediate results is needed in a different context.

  → **maintenance nightmare** – If the result is needed in another context, add a corresponding operation explicitly to the interface.

Yet with today's hardware and programming languages, this is hardly an issue any more; at the time of (Parnas, 1972), it clearly was.
A Classification of Modules (Nagl, 1990)

- **functional modules**
  - group computations which belong together logically,
  - do not have “memory” or state, that is, behaviour of offered functionality does not depend on prior program evolution,
  - Examples: mathematical functions, transformations

- **data object modules**
  - realise encapsulation of data,
  - a data module hides kind and structure of data, interface offers operations to manipulate encapsulated data
  - Examples: modules encapsulating global configuration data, databases

- **data type modules**
  - implement a user-defined data type in form of an abstract data type (ADT)
  - allows to create and use as many exemplars of the data type
  - Example: game object

- In an object-oriented design,
  - classes are data type modules,
  - data object modules correspond to classes offering only class methods or singletons (→ later),
  - functional modules occur seldom, one example is Java’s class Math.

---

**Example**

(i) information hiding and data encapsulation not enforced.
(ii) → negative effects when requirements change.
(iii) enforcing information hiding and data encapsulation by modules.
(iv) abstract data types.
(v) object oriented without information hiding and data encapsulation,
(vi) object oriented with information hiding and data encapsulation.
Example: Module ‘List of Names’

- **Task**: store a list of names in \( N \) of type “list of string”.

- **Operations**: (in interface of the module)
  
  - **insert** (string \( n \)):
    
    - **pre-condition**:
      \[ N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_m, m \in \mathbb{N}_0, \forall 0 \leq j < m \implies n_j < n_{j+1} \]
    
    - **post-condition**:
      \[ N = n_0, \ldots, n_{i-1}, n_{i+1}, n, n_{i+1}, \ldots, n_m, \forall 0 \leq j < m \implies n_j < n_{j+1} \]

  - **remove** (int \( i \)):
    
    - **pre-condition**:
      \[ N = n_0, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_m, m \in \mathbb{N}_0, 0 \leq i < m \]
    
    - **post-condition**:
      \[ N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_m \]

  - **get** (int \( i \)) : string;
    
    - **pre-condition**:
      \[ N = n_0, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_m, m \in \mathbb{N}_0, 0 \leq i < m \]
    
    - **post-condition**:
      \[ N = \text{old}(N), \text{retval} = n_i \]

  - **dump**();
    
    - **pre-condition**:
      \[ N = n_0, \ldots, n_m, m \in \mathbb{N}_0 \]
    
    - **post-condition**:
      \[ N = \text{old}(N) \]
    
    - **side-effect**: \( n_0, \ldots, n_m \) printed to standard output in this order.

A Possible Implementation: Plain List, no Duplicates

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

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

void insert(std::string n) {
    std::vector<std::string>::iterator it =
        lower_bound(names.begin(), names.end(), n);
    if (it == names.end() || *it != n)
        names.insert(it, n);
}

void remove(int i) {
    names.erase(names.begin() + i);
}

std::string get(int i) {
    return names[i];
}

int main() {
    insert("Berger");
    insert("Schulz");
    insert("Neumann");
    insert("Meyer");
    insert("Wernersen");
    insert("Neumann");
    dump();
    remove(1);
    insert("Mayer");
    dump();
    names[2] = "Naumann";
    dump();
    return 0;
}
```

Output:

```
Berger
Meyer
Neumann
Schulz
Wernersen
```

access is bypassing the interface — no problem, so far
Change Interface: Support Duplicate Names

- **Task:** in addition, \( \text{count}(n) \) should tell how many \( n \)'s we have.

- **Operations:** (in interface of the module)
  - `insert(string n);`
    - **pre-condition:**
      \[ N = n_0, \ldots, n_i, \ldots, n_{m-1}, m \in \mathbb{N}_0, \forall 0 \leq j < m \implies n_j < n_{j+1} \]
    - **post-condition:**
      \[ \text{if } n_i < n_{i+1} \land N = n_0, \ldots, n_i, \ldots, n_{m-1}, \text{count}(n) = 1 \]
      \[ \text{if } n_i = n_j \text{ for some } 0 \leq i < m, N = \text{old}(N), \text{count}(n) = \text{old(count(n))} + 1 \]
  - `remove(int i);`
    - **pre-condition:**
      \[ N = n_0, \ldots, n_{i-1}, n_i, n_{i+1}, \ldots, n_{m-1}, m \in \mathbb{N}_0, 0 \leq i < m \]
    - **post-condition:**
      \[ \text{if } \text{count}(n_i) = 1, N = n_0, \ldots, n_{i-1}, n_{i+1}, \ldots, n_{m-1} \]
      \[ \text{if } \text{count}(n_i) > 1, N = \text{old}(N), \text{count}(n_i) = \text{old(count(n_i))} - 1. \]
  - `get(int i): string;` and `dump();`
    - unchanged contract

Changed Implementation: Support Duplicates

```cpp
1 std::vector<int> count;
2 // std::vector<std::string> names;
3 void insert(std::string n) { // names.begin(), n; }
4 std::vector<std::string>::iterator
5 it = lower_bound(names.begin(), n);
6 // insert(it, n);
7 count.insert(count.end(), 1);
8 }
9 if (it != names.end()) {
10   names.insert(it, n);
11   count.insert(count.end(), 1);
12 } else {
13   if (*it == n) { // count.begin() +
14     count.insert(count.begin() +
15                  // count.begin() +
16                  // it - names.begin());
17     names.insert(it, n);
18   } else { // count.begin() +
19     ++(*count.begin() +
20                  // it - names.begin());
21   }
22 }
23 }
24 void remove(int i) { // [--count[1] += 0];
25   // names.erase(names.begin(i + 1));
26   count.erase(count.begin(i + 1));
27 }
28 std::string get(int i) { // names[i];
29   return names[i];
30 }
31 }
32
33 1 int main() { // Berger
34   insert("Berger");
35   insert("Schulz");
36   insert("Meyer");
37   insert("Wernersen");
38   insert("Neumann");
39   dump();
40   remove(1);
41   insert("Mayer");
42   dump();
43   // names[2] = "Neumann"
44   dump();
45   return 0;
46 } // Berger
47 // Mayer
48 // Schulz
49 // Wernersen
50 // Neumann
```

Output:

- Berger: 1
- Mayer: 1
- Neumann: 2
- Schulz: 1
- Wernersen: 1

access is bypassing the interface – and corrupts the data-structure
# Data Encapsulation + Information Hiding

```cpp
#include <string>
#include <iostream>
#include <vector>
#include "mod_deih.h"

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

void insert( std::string n ) {
    count.push_back( 0 );
    names.push_back( n );
}

void remove( int i ) {
    if ( --count[i] == 0 ) {
        names.erase( names.begin() + i );
        count.erase( count.begin() + i );
    }
}

std::string get( int i ) {
    return names[i];
}

int main() {
    insert( "Berger" );
    insert( "Schulz" );
    insert( "Neumann" );
    insert( "Wernersen" );
    insert( "Neumann" );
    dump();
    remove( 1 );
    insert( "Mayer" );
    dump();
}
```

**Header**

```cpp
#include string
```

**Source**

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
#include "mod_deih.h"

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

void insert( std::string n ) {
    count.push_back( 0 );
    names.push_back( n );
}

void remove( int i ) {
    if ( --count[i] == 0 ) {
        names.erase( names.begin() + i );
        count.erase( count.begin() + i );
    }
}

std::string get( int i ) {
    return names[i];
}
```

**Output:**

```
Berger: 1
Mayer: 1
Neumann: 2
Schulz: 1
Wernersen: 1
```
Abstract Data Type

```cpp
#include <string>

typedef void* Names;

Names new_Names();

void dump( Names names );

void insert( Names names, std::string n );

void remove( Names names, int i );

std::string get( Names names, int i );

#include "mod_adt.h"

typedef struct {
  std::vector<int> count;
  std::vector<std::string> names;
} implNames;

Names new_Names() {
  return new implNames;
}

void insert( Names names, std::string n ) { 
  implNames* in = (implNames*)names;
  std::vector<std::string>::iterator it = lower_bound( in->names.begin(), in->names.end(), n );
  if ( it == in->names.end() ) {
    in->names.insert( it, n );
  }
}

int main() {
  Names names = new_Names();
  insert( names, "Berger" );
  insert( names, "Schöla" );
  insert( names, "Neumann" );
  insert( names, "Meyer" );
  insert( names, "Wemersen" );
  insert( names, "Neumann" );
  dump( names );
  remove( names, 1 );
  insert( names, "Mayer" );
  dump( names );
  names[2] = "Naumann";
  if (ndef AVOID_PROBLEM) { 
    remove( names, 2 );
    insert( names, "Naumann" );
  }
  dump( names );
  return 0;
}
```

mod_adt_main.cpp: In function 'int main()':
mod_adt_main.cpp:22:10: warning: pointer of type 'void*' used in arithmetic [-Wpointer-arith]
mod_adt_main.cpp:22:10: error: 'Names (aka void*)' is not a pointer-to-object type
Abstract Data Type

```cpp
#include <string>

namespace Names { 

    template <typename T>
    class Names 
    { 
        public:

            Names(); 
            ~Names();
            std::vector<std::string> names; 

            void insert(const T& n);
            void remove(int i);

        private:

            std::string get(int i); 
    };

};
```

Object Oriented

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

namespace Names { 

    class Names 
    { 
        public:

            Names(); 
            ~Names();
            std::vector<std::string> names; 

            void insert(const std::string& n);
            void remove(int i);

        private:

            std::string get(int i); 
    };

};
```
Object Oriented

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

struct Names {
  std::vector<std::string> names;

  void dump();
  void insert(std::string n);
  void remove(int i);

  std::string get(int i);
};

int main() {
  Names names;
  names.insert("Werner");
  names.insert("Mayer");
  names.insert("Naumann");
  names.insert("Neumann");
  names.insert("Berg");
  names.insert("Schulz");
  names.insert("Schul");
  names.insert("Werners");
  return 0;
}
```

Output:

```
Berger: 1
Mayer: 1
Neumann: 2
Schul: 1
Werners: 1
```

Object Oriented + Data Encapsulation / Information Hiding

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

class Names {
  private:
    std::vector<int> count;
    std::vector<std::string> names;

  public:
    Names();
    void dump();
    void insert(std::string n);
    void remove(int i);

    std::string get(int i);
};

int main() {
  Names names;
  names.insert("Werner");
  names.insert("Mayer");
  names.insert("Naumann");
  names.insert("Neumann");
  names.insert("Berg");
  names.insert("Schulz");
  names.insert("Schul");
  names.insert("Werners");
  return 0;
}
```

In file included from mod_oo_deih_main.h:10:

```
mod_oo_deih.h: In function 'int main()':
mod_oo_deih.h:9:28: error: std::vector<std::basic_string<char>> Names::names is private
```

In file included from mod_oo_deih_main.cpp:22:

```
mod_oo_deih.cpp: error: within this context
```

Object Oriented

```cpp
#include "mod_oo_deih.h"

int main() {
  Names names = new Names();
  names.insert("Werner");
  names.insert("Mayer");
  names.insert("Naumann");
  names.insert("Neumann");
  names.insert("Berg");
  names.insert("Schulz");
  names.insert("Schul");
  names.insert("Werners");
  return 0;
}
```

```
Berger: 1
Mayer: 1
Neumann: 2
Schul: 1
Werners: 1
```

access is bypassing the interface – and corrupts the data-structure
“Tell Them What You’ve Told Them”

(i) information hiding and data encapsulation not enforced.

(ii) negative effects when requirements change.

(iii) enforcing information hiding and data encapsulation by modules.

(iv) abstract data types.

(v) object oriented without information hiding and data encapsulation.

(vi) object oriented with information hiding and data encapsulation.
Software Modelling
**Model**

Definition. [Folk] **A model** is an abstract, formal, mathematical representation or description of structure or behaviour of a (software) system.

Definition. (Glinz, 2008, 425)

A **model** is a concrete or mental **image** (Abbild) of something or a concrete or mental **archetype** (Vorbild) for something.

Three properties are constituent:

(i) the **image attribute** (Abbildungsmerkmal), i.e. there is an entity (called **original**), whose image or archetype the model is.

(ii) the **reduction attribute** (Verkürzungsmerkmal), i.e. only those attributes of the original that are relevant in the modelling context are represented.

(iii) the **pragmatic attribute**, i.e. the model is built in a specific context for a specific **purpose**.

---

**Example: Process Model**

*From Building Blocks to Process (And Back)*

[Diagram showing process model building blocks and planning process.]

---

*From Building Blocks to Process (And Back)*

[Diagram showing process model building blocks and implementation process.]

---

*From Building Blocks to Process (And Back)*

[Diagram showing process model building blocks and testing process.]

---

*From Building Blocks to Process (And Back)*

[Diagram showing process model building blocks and integration process.]
### Example: Design-Models in Construction Engineering

#### 1. Requirements
- Shall fit on given piece of land.
- Each room shall have a door.
- Furniture shall fit into living room.
- Bathroom shall have a window.
- Cost shall be in budget.

#### 2. Designmodel

#### 3. System

**Observation (1):** Floorplan abstracts from certain system properties, e.g.,
- kind, number, and placement of bricks,
- subsystem details (e.g., window style),
- water pipes/wiring, and
- wall decoration

→ architects can efficiently work on appropriate level of abstraction

**Observation (2):** Floorplan preserves/determines certain system properties, e.g.,
- house and room extensions (to scale),
- presence/absence of windows and doors,
- placement of subsystems (such as windows).

→ find design errors before building the system (e.g., bathroom windows)
Views and Viewpoints

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

IEEE 1471 (2000)

**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)

(Ludewig and Lichter, 2013):

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

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

**logical view**

**process 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 (→ later) – so no need for (extensive) particular documentation.
**Structure vs. Behaviour**

- **Form of the states** in $\Sigma$ (also actions $A$): 
  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.”
  $\rightarrow$ 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.”
  $\rightarrow$ 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.
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.

An Outlook to UML
A Brief History of the Unified Modelling Language (UML)

- Boxes/lines and finite automata are used to visualise software for ages.

- **1970’s, Software Crisis**—
  - Idea: learn from engineering disciplines to handle growing complexity.
  - Modelling languages: Flowcharts, Nassi-Shneiderman, Entity-Relation Diagrams

- **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)
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)
- Object-Oriented Software Engineering (OOSE) (Jacobson et al., 1992)

Each “persuasion” selling books, tools, seminars…

Late 1990’s: joint effort of “the three amigos” UML 0.x and 1.x Standards published by Object Management Group (OMG), “international, open membership, not-for-profit computer industry consortium”. Much criticised for lack of formality.

Since 2005: UML 2.x, split into infra- and superstructure documents.
**UML Overview (OMG, 2007, 684)**

![Diagram of UML Overview]

**Figure A.5 - The taxonomy of structure and behavior diagram**

**Dobing and Parsons (2006)**

---

**Topic Area Architecture & Design: Content**

- **Introduction and Vocabulary**
- **Principles of Design**
  1. Modularity
  2. Separation of concerns
  3. Information hiding and data encapsulation
  4. Abstract data types, object orientation

- **Software Modelling**
  1. Views and viewpoints, the 4+1 view
  2. Model-driven/-based software engineering
  3. Unified Modelling Language (UML)
  4. Modelling structure
    a. Simplified class diagrams
    b. Simplified object diagrams
    c. Simplified object constraint logic (OCL)
  5. Modelling behaviour
    a. Communicating finite automata
    b. Upaal query language
    c. Basic state-machines
    d. An outlook on hierarchical state-machines

- **Design Patterns**
Tell Them What You’ve Told Them...

- **Design** structures a system into **manageable units**.
- **Principles of (Good) Design:**
  - modularity, separation of concerns,
  - information hiding / data encapsulation
- **Model:** a concrete or mental **image** or archetype with
  - image attribute,
  - reduction attribute,
  - pragmatic attribute.
  here: abstract, formal, mathematical description.
- **Software Modelling:** views and viewpoints, e.g. 4+1
- **Model-driven** Software Engineering
- **Unified Modelling Language:**
  - a family of **modelling languages**.

References


