UML models: Object-orientation and Design Patterns

Software Engineering Theory

Kristian Sandahl
Department of Computer and Information Science
2014-09-16
Requirements

System Design
(Architecture, High-level Design)

Module Design
(Program Design, Detailed Design)

Implementation of Units (classes, procedures, functions)

Unit testing

Acceptance Test
(Release testing)

System Testing
(Integration testing of modules)

Module Testing
(Integration testing of units)

Validation, Verification, System Design, Module Design, Implementation

Project Management, Software Quality Assurance (SQA), Supporting Tools, Education

Maintenance
The goals of module design, again

• Provide the expected function

• Prepare for change:
  • Separation of concern
  • Testability
  • Understandability

• Contribute to quality, eg:
  • Performance
  • Usability
  • Reliability
  • ...

• Map for the implementers and testers
Well-known Diagrams of UML

- UML 2.5 Diagram
  - Structure Diagram
    - Class Diagram
    - Object Diagram
  - Behavior Diagram
    - Use-Case Diagram
      - State Machine Diagram
      - Interaction Diagram
    - Sequence Diagram
  - Deployment Diagram
    - Package Diagram
    - Component Diagram
Well-known Diagrams of UML
Instance models

Specific

Kristian: CoffeeCustomer buys Cup1: CupOfCoffee makes IDA-B-house: Machine

Generic

aCoffeeCustomer: CoffeeCustomer buys aCup: CupOfCoffee makes aMachine: Machine

Short hand

: CoffeeCustomer buys : CupOfCoffee makes : Machine

Related: Roles

: CoffeeCustomer buys : CupOfCoffee makes : Machine
State machine diagram

For class CoinHandler:

- **checking**
  - trigger event, causing transition
  - action, reaction to the event
  - state
  - transition

- **idle**
  - this object
  - start state marker

- **insertCoin()/checkCoin(self)**

- **falseCoin()/returnCoin(self)**
A few more states

Kristian’s alarm clock starts sounding at 6.00 with a nasty signal. He can now do either of three things:

a) Turn the alarm off;
b) Press the snooze button; or

c) Do nothing. If the snooze button is pressed the signal will turn off and start sounding after 5 minutes again.

When an hour has passed from the first time the alarm sound started, the snooze button has no effect.

After that the alarm sound starts, the signal will last for 2 minutes. If no action has been taken during these 2 minutes, the absence of action will have the same effect as if the snooze button were pressed exactly when the alarm stopped to sound.
Orthogonal, composite states

course attempt

state machine

Studying

Lab 1

lab1 done

Lab 2

lab2 done

Project

project done

Final exam

pass

fail

Failed

Passed

orthogonal state

orthogonal region
Explicit exit points

course attempt

Studying

Lab 1 \(\xrightarrow{\text{lab1 done}}\) Lab 2 \(\xrightarrow{\text{lab2 done}}\)

Project \(\xrightarrow{\text{project done}}\)

Final exam \(\xrightarrow{\text{pass}}\)

fail

failed \(\xrightarrow{\text{failed}}\) passed \(\xrightarrow{\text{passed}}\)
Activity diagram ≠ State diagram

Initial node

Activity:
1. insert coin
2. decision: coin accepted?
   - [yes]: fork
   - [no]: final node
3. fork:
   - brew coffee
   - add sugar/whitener
4. join:
   - add hot water to adjust strength
   - pour coffee

Final node
Sequence diagram

- role: CoffeeCustomer
- Message (synchronous)
  - insertCoin
  - machineReady
  - pressButton(b1)
  - pourCoffee
- Life line of object
- Procedure is active
- Procedure is active
Sequence diagram with several objects

: CoffeeCustomer

: Interface

: CoinHandler

: Brewer

- insertCoin
- transport
- coinAccepted
- warmUp
- makeOrder(o1)
- pourCoffee

Timing constraint

\{ 0 < 5s \}

litIndicators

pressButton(b1)

pourCoffee

Return message
Combining fragments of sequence diagrams

SD processOrder

:Order

:TicketDB

:Account

create

ref

Get existing customer data

[get next item]

reserve(date,no)

add(seats)

answer

destruction

loop condition

loop

loop

gate
Combining fragments of sequence diagrams

SD Get existing customer data

:Order

create

:TicketDB

Getdata(c1)

:Account

Data(c1)
Combining fragments of sequence diagrams

SD processOrder

:Order

create

:TicketDB

ref

Get existing customer data

:Account

loop

[get next item]

reserve(date,no)

add(seats)

answer

destruction

loop

loop condition
More fragments of sequence diagrams

```
:Order

loop

[get next item]

reserve(date,no)

alt

[available]

add(seats)

reject

[unavailable]

:TicketDB

```

- **Guard condition**
- **Nested conditional**
- **Alternate branches**
The Unified Modeling Language

The Unified Modeling Language™ (UML®) is a standard visual modeling language intended to be used for
• modeling business and similar processes,
• analysis, design, and implementation of software-based systems

UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.

UML can be applied to diverse application domains (e.g., banking, finance, internet, aerospace, healthcare, etc.) It can be used with all major object and component software development methods and for various implementation platforms (e.g., J2EE, .NET).

UML is a standard modeling language, not a software development process. UML 1.4.2 Specification explained that process:
• provides guidance as to the order of a team’s activities,
• specifies what artifacts should be developed,
• directs the tasks of individual developers and the team as a whole, and
• offers criteria for monitoring and measuring a project’s products and activities.

UML is intentionally process independent and could be applied in the context of different processes. Still, it is most suitable for use case driven, iterative and incremental development processes. An example of such process is Rational Unified Process (RUP).

UML is not complete and it is not completely visual. Given some UML diagram, we can’t be sure to understand depicted part or behavior of the system from the diagram alone. Some information could be intentionally omitted from the diagram, some information represented on the diagram could have different interpretations, and some concepts of UML have no graphical notation at all, so there is no way to depict those on diagrams.

For example, semantics of multiplicity of actors and multiplicity of use cases on use case diagrams is not defined precisely in the UML specification and could mean either concurrent or successive usage of use cases.

Name of an abstract classifier is shown in italics while final classifier has no specific graphical notation, so there is no way to determine whether classifier