Softwaretechnik

Lecture 04: Object-Oriented Analysis

Peter Thiemann

University of Freiburg, Germany

SS 2011

Object-Oriented Analysis

- After introduction of OOP: need for OOA and OOD
- Purpose: Building OO models of software systems
- No generally accepted methodology; many different approaches: Booch, Rumbaugh (OMT), Coad/Yourdon, Jacobson (OOSE), Wirfs-Brock, . . .
- Current approaches rely on **UML** (Unified Modeling Language, Booch/Jacobson/Rumbaugh)
- UML supports many kinds of semi-formal modeling techniques
  - **use case diagrams**
  - **class diagrams**
  - **sequence diagrams**
  - **statechart diagrams**
  - **activity diagrams**
  - **deployment diagrams**

The Concept “Model”

(according to Herbert Stachowiak, 1973)

Representation

A model is a representation of an original object.

Abstraction

A model need not encompass all features of the original object.

Pragmatism

A model is always goal-oriented.

- Modeling creates a representation that only encompasses the relevant features for a particular purpose.

Variations of Models

Informal models

- informal syntax, intuitive semantics
- ex: informal drawing on blackboard, colloquial description

Semi-formal models

- formally defined syntax (metamodel), intuitive semantics
- ex: many diagram types of UML

Formal models

- formally defined syntax and semantics
- ex: logical formulae, phrase structure grammars, programs

Class Diagram (UML)

- Data-oriented view, cf. ERD
- Representation of **classes** and their **static relationships**
- No information on dynamic behavior
- Notation is graph with
  - **nodes**: classes (rectangles)
  - **edges**: various relationships between classes
- May contain interfaces, packages, relationships, as well as instances (objects, links)

Classes

A class box has compartments for

- Class name
- Attributes (variables, fields)
- Operations (methods)

- only name compartment obligatory
- additional compartments may be defined
- class (static) attributes / operations underlined

Relations Between Classes

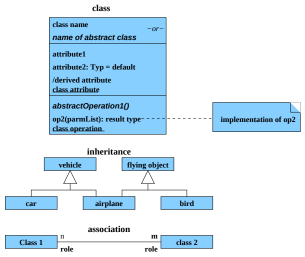
Binary Association

- indicates “collaboration” between two classes (possibly reflexive)
- solid line between two classes
- optional:
  - association name
  - decoration with role names
  - navigation (Design)
  - multiplicities (Design)

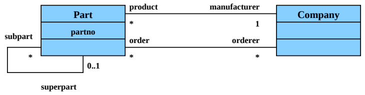
Generalization

- indicates subclass relation
- solid line with open arrow towards super class

Example Class Diagram



Example Class Diagram



Ten Steps Towards an OOA Model

Heide Balzert

1. Data analysis: identify classes
2. Identify associations and compositions
3. Identify attributes and operations for each class
4. Construct object life cycle
5. Introduce inheritance
6. Identify internal operations
7. Specify operations
8. Check inheritance
9. Check associations and compositions
10. Decompose in subsystems

Step: Identify Classes

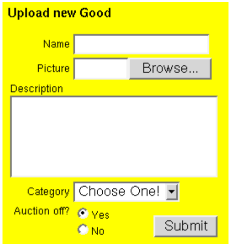
- identify tangible entities: physical objects (airplane), roles (manager), events (request, form), interactions (meeting), locations (office), organizational units (company)
- top-down: scan verbal requirements
  - nouns → objects, attributes
  - verbs → operations

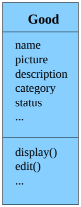
  bottom-up:
  - collect attributes (data) and operations
  - combine into classes
- name of class: concrete noun, singular, describes all objects (no roles)
- classes related via invariable 1:1 associations may be joined

Step: Identify Associations and Compositions

- permanent relations between objects
- scan verbal requirements for verbs
- technical subsidiarity: composition
- communication between objects → association
- determine roles
- snapshot / history required?
- constraints?
- are there attributes / operations for association?
- determine cardinalities

Attributes and Operations by Form Analysis



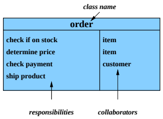


Step: Identify Attributes and Operations

CRC Cards (Wirfs-Brock)

- CRC = Class-Responsibility-Collaboration
- initially, a class is assigned **responsibilities** and **collaborators**
- collaborator is a class cooperating to fulfil responsibilities
- **three-four** responsibilities per card (class); otherwise: split class
- developed iteratively through series of meetings

Example CRC Card



Classes From Use Cases

Use Case: buy product

- Locate product in catalogue
- Browse features of product
- Place product in shopping cart
- Proceed to checkout
- Enter payment info
- Enter shipping info
- Confirm sale

F# Notation for Datatypes

```
type sale =         { cart: shoppingCart;
                      shipment: shipmentInfo;
                      payment: paymentInfo }
and  shoppingCart = { contents: product list }
and  shipmentInfo = { name: string;
                      address: string }
and  paymentInfo =  { accountNr: string;
                      bankingCode: string }
and  product =      { name: string;
                      price: int;
                      features: feature list }
and  feature =      { name: string }
```

- Named record types

Classes from Requirements

A graphics program should draw different geometric shapes in a coordinate system. There are four kinds of shapes:

- *Rectangles given by upper left corner, width, and height*
- *Disks given by center point and radius*
- *Points*
- *Overlays composed of two shapes*

Classes from Requirements

```
type cartPt = { x: int; y: int }
and shape =
     Rectangle of rectangle
   | Disk of disk
   | Point of point
   | Overlay of overlay
and rectangle = { loc: cartPt; width: int; height: int }
and disk = { loc: cartPt; radius: int }
and point = { loc: cartPt }
and overlay = { lower: shape; upper: shape }
```

- Sum type (shape) for alternatives

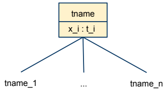
Mapping from F# Types to Class Diagrams

Mapping a type definition

$$[\![\text{type } tdef_1 \text{ and } \dots \text{ and } tdef_n]\!] = [\![tdef_1]\!] \cup \dots \cup [\![tdef_n]\!]$$

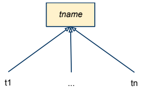
Mapping a record type

$[\![\text{tname} = \{x_i : t_i, y_j : \text{tname}_j\}]\!] =$

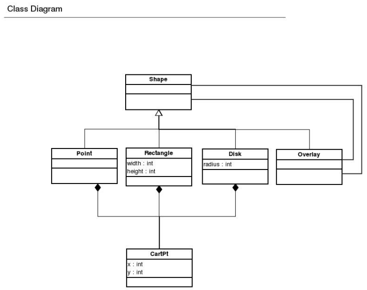


Mapping a sum type

$[\![\text{tname} = T_1 \text{ of } t_1 \mid \dots \mid T_n \text{ of } t_n]\!] =$



Applied to Example Code

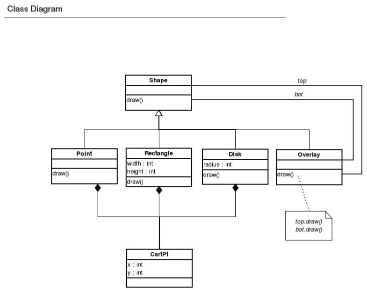


. . . Operations

A graphics program should draw different geometric shapes . . .

- Each class should have a draw() operation
- Shape should also have draw() operation
- Discovered the “Composite Pattern”!

Example Code with Draw Method



Step: Construct Object Life Cycle

Object Life Cycle

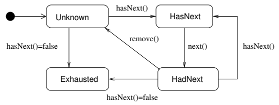
- Object creation
- Initialization
- . . .
- Finalization
- Object destruction

Life Cycle — Type State

- operations can only be executed in particular state
- idea: incoming message (in class diagram) $\hat{=}$ event (in a **statechart diagram**) that triggers the operation

Example: Java Iterator — Statechart Diagram

```
interface Iterator<E> {
  /** Returns true if the iteration has more elements. */
  public boolean hasNext();
  /** Returns the next element in the iteration. */
  public E next();
  /** Removes from the underlying collection the last element
      returned by the iterator (optional operation). */
  public void remove();
}
```



Statechart Diagram

- Modeling the evolving state of an object
- Based on deterministic finite automaton (FSA) $A = (Q, \Sigma, \delta, q_0, F)$ where
  - Q: finite set of states
  - Σ: finite input alphabet
  - δ: $Q \times \Sigma \longrightarrow Q$ transition function
  - $q_0 \in Q$ initial state
  - $F \subseteq Q$ set of final states

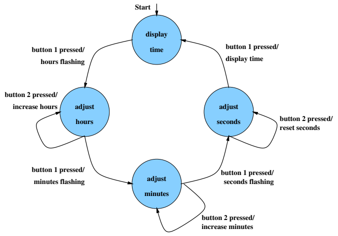
Graphical Representation of FSA

- **nodes:** states of the automaton (circles or rectangles)
- arrow pointing to q_0
- final states indicated by double circle
- **edges:** if $\delta(q, a) = q'$ then **transition** labeled a from q to q'

FSA with output specifies a translation $\Sigma^* \to \Delta^*$

- $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$
- replace final states F by output alphabet Δ and output function λ
- **Mealy-automaton:** $\lambda : Q \times \Sigma \longrightarrow \Delta$
  edge from q to $\delta(q, a)$ additionally carries $\lambda(q, a)$
- **Moore-automaton:** $\lambda : Q \longrightarrow \Delta$
  state q labeled with $\lambda(q)$

Example: Digital Clock as a Mealy-automaton

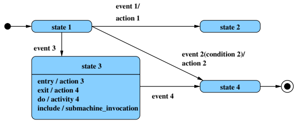

Start
display time
button 1 pressed/ hours flashing
button 1 pressed/ display time
button 2 pressed/ increase hours
adjust hours
adjust seconds
button 2 pressed/ reset seconds
button 1 pressed/ minutes flashing
adjust minutes
button 1 pressed/ seconds flashing
button 2 pressed/ increase minutes

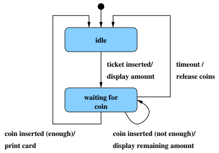
Drawback: FSAs get big too quickly $\to$ structuring required

Statechart Diagram (Harel, UML)

- hybrid automata ("Moore + Mealy")
- each state may have
  - **entry action:** executed on entry to state ≅ labeling all incoming edges
  - **exit action:** executed on exit of state ≅ labeling all outgoing edges
  - **do activity:** executed while in state
- composite states
- states with history
- concurrent states
- optional: conditional state transitions

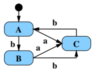
Example: Statechart Diagram

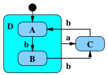


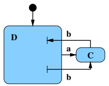


Composite States

- states can be grouped into a composite state with designated start node (→ hierarchy)
- edges may start and end at any level
- transition from a composite state ≅ set of transitions with identical labels from all members of the composite state
- transition to a composite state leads to its initial state
- transitions may be "stubbed"

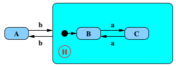






States with History

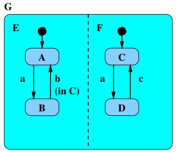
- composite state with history — marked **(H)** — remembers the internal state on exit and resumes in that internal state on the next entry



- the history state indicator may be target of transitions from the outside and it may indicate a default "previous state"
- "deep history" **(H*)** remembers nested state

Concurrent States

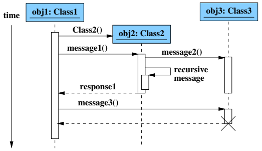
- composite state may contain **concurrent state regions** (separated by dashed lines)
- all components execute concurrently
- transitions may depend on state of another component (synchronisation)
- explicit synchronization points
- concurrent transitions



sequence of states on input abcb:
(A, C), (B, D), (B, D), (B, C), (A, C)

Alternative: Sequence Diagram

- description of the sequence of messages
- $\rightarrow$ communications protocols



Step: Introduce Inheritance

- Use sparingly!
- Use inheritance for abstracting common patterns:
  Collect common attributes and operations in abstract superclass
- Alternative: collect in separate class and use composition

Step: Specify Operations

- Data-driven development: [Jackson]
  Derive structure of operation from data it operates on
- Test-driven development: [Beck]
  Specify a set of meaningful test cases
- Design by contract: [Meyer]
  - Define class invariants
  - Specify operations by pre- and postconditions
- Pseudocode Programming Process (PPP): [McConnell]
  - Start with high-level pseudocode
  - Refine pseudocode until implementation obvious