

Software Design

Fritz Henglein
DIKU

March 6, 2003

DIKU Project Seminar

Sources

- ✍ Martin Fowler, Kendall Scott, "UML Distilled," 2nd ed., Addison-Wesley, 1999
- ✍ **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994**
- ✍ Kevin Lano, Jose Luiz Fiadeiro, Luis Andrade, "Software Design Using Java 2", Palgrave Macmillan, 2002

March 6, 2003

DIKU Project Seminar

Outline

- ✍ Software design: What's that?
- ✍ Design granularity: architecture, subsystem, module, unit
- ✍ Generic design techniques
- ✍ UML
- ✍ Design patterns

March 6, 2003

DIKU Project Seminar

What is software design? Definition of...

- ✍ **Organization** of a software system into modules/components/classes or other units
- ✍ **Behavior** and **responsibilities** of units
- ✍ **Interactions** and **collaborations** between units

March 6, 2003

DIKU Project Seminar

UML: Unified Modeling Language

- ✍ Drawing notations for supporting software design process and conveying designs
- ✍ "Bubbles and arcs":
 - ✍ Meant to be used informally and selectively;
 - ✍ No unique, definite semantics.
- ✍ CASE tool: drawing tool, plus more; e.g., automatic code generation (Fowler: Visio + MS Word = gets you far)

March 6, 2003

DIKU Project Seminar

Rational Unified Process

1. **Inception:** Business rationale, scope
2. **Elaboration:** Requirements, high-level analysis and design of baseline architecture, construction plan, risk management
3. **Construction**
 - ✍ iteration 1: use cases A,B,C implemented
 - ✍ iteration 2: use cases D,E implemented
 - ✍ ...
4. **Transition:** Beta testing, performance tuning, user training.

March 6, 2003

DIKU Project Seminar

UML Notations

- ∞ Use cases
- ∞ Class diagrams
- ∞ Interaction diagrams:
 - ∞ Sequence diagrams
 - ∞ Collaboration diagrams
- ∞ State diagrams
- ∞ Activity diagrams
- ∞ Physical diagrams (no more on those)

March 6, 2003

DIKU Project Seminar

General design techniques

- ∞ Design **testability** and systematic test into the software ("self-testing software")
- ∞ **Refactoring**: redesigning ("cleaning up") existing code without adding functionality
- ∞ Use **design patterns**: Fosters reuse, communication and process and result predictability
- ∞ Recommendations:
 - ∞ Don't combine refactoring with adding functionality
 - ∞ Refactor rather often and in small doses

March 6, 2003

DIKU Project Seminar

Use cases

- ∞ **Scenario**: sequence of steps describing interaction between user and system
- ∞ **Use case**: set of scenarios tied together by a common user goal
- ∞ **Use case notation**:
 - ∞ Name (user goal)
 - ∞ Sequence of numbered steps (main scenario)
 - ∞ Collection of alternatives (alternative scenarios)
- ∞ **Use case diagram**: stick man drawings

March 6, 2003

DIKU Project Seminar

Use cases...

- ∞ Present *external view* of a system ('what', not 'how')
- ∞ Used for:
 - ∞ Requirements capture (during elaboration)
 - ∞ Planning and controlling iterative construction phase

March 6, 2003

DIKU Project Seminar

Class Diagrams

- ∞ **Class diagram**: describes types of *objects* in a system and their static *relationships*
- ∞ **Class**: node (box) showing attributes and operations of a class
- ∞ Relationships:
 - ∞ **Association**: analogous to relationships in entity-relationship model. ("A customer rents videos.")
 - ∞ **Subtype**: conceptual subset relation ("A nurse is a person.")

March 6, 2003

DIKU Project Seminar

Perspectives

- ∞ Class diagrams are used in different stages and with different *perspectives*:
 - ∞ **Conceptual** (for domain modeling): Concepts in the domain under study and their relations (think 'things out there', not software)
 - ∞ **Specification** (for software architecture): Description of class interfaces and class relationships (interfaces and constraints only, no concrete code)
 - ∞ **Implementation** (for code description): Description of implementation (think code).

March 6, 2003

DIKU Project Seminar

Constraint rules

- ≠ Express class invariants (invariants that must always hold: after initialization, and after each method call)
- ≠ Design by Contract:
 - ≠ Pre- and postconditions for method invocations
 - ≠ Class invariants
 - ≠ Specify important semantic properties, not expressible in underlying programming language type system

March 6, 2003

DIKU Project Seminar

Class diagrams...

- ≠ Used for:
 - ≠ Domain modeling (elaboration)
 - ≠ Construction planning (detailed design)
 - ≠ Coding (implementation description)
- ≠ CRC cards:
 - ≠ class-responsibility-collaboration
 - ≠ used for domain modeling (arriving at a conceptual class model)

March 6, 2003

DIKU Project Seminar

Sequence diagrams

- ≠ Description of execution behavior (message sends and receives) among a (fixed, finite) collection of objects
- ≠ Each object is shown with its own thread of control, even if execution is sequential (with send-receive = call-return)

March 6, 2003

DIKU Project Seminar

Collaboration diagrams

- ≠ Different way of presenting sequence diagrams

March 6, 2003

DIKU Project Seminar

State diagrams

- ≠ **State diagram:**
 - ≠ shows state transitions of single object as (finite) state machine
 - ≠ transitions may be conditional
- ≠ Used to describe 'life cycle' of single object for objects with complex stateful behavior

March 6, 2003

DIKU Project Seminar

Activity diagrams

- ≠ Graphical notation for join/fork calculus
- ≠ Used for:
 - ≠ displaying (concurrent) execution of a collection of objects,
 - ≠ "orchestration" of objects (workflow, mediator pattern)
 - ≠ flow chart representation of sequential program
- ≠ Activity diagrams with swimlanes: activities are mapped to objects (object = particular swimlane)

March 6, 2003

DIKU Project Seminar

How to use UML

- ☞ Use to support software development process for communication across teams (same time) and across time (same person)
- ☞ “A picture is worth a thousand words.”

March 6, 2003

DIKU Project Seminar

Patterns

- ☞ “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution a million times over, without ever doing it the same way twice”
Christopher Alexander, 1977

March 6, 2003

DIKU Project Seminar

Design pattern (GoF)

- ☞ Description of communicating objects and classes that are customized to solve a general design problem in a particular context
- ☞ Names, abstracts, identifies key aspects of a common design structure that makes it useful for creating a reusable object-oriented design.
- ☞ Bigger than single class, smaller than whole framework/application

March 6, 2003

DIKU Project Seminar

Design pattern elements

- ☞ **Pattern name:** Handle for describing design problem
- ☞ **Problem:** When to apply pattern. Explains problem and its context.
- ☞ **Solution:** Elements making up design (relationships, responsibilities, collaborations)
- ☞ **Consequences:** Results and trade-offs of applying pattern (implementation issues, performance, reuse etc.)

March 6, 2003

DIKU Project Seminar

Design pattern description

- ☞ **Name:** Essence in one word
- ☞ **Intent:** Short statement
- ☞ **Also known as:** Alias
- ☞ **Motivation:** concrete scenario
- ☞ **Applicability:** Situations for application
- ☞ **Structure:** Graphical representation of design (UML)
- ☞ **Participants:** Classes and objects
- ☞ **Collaborations:** How participants collaborate
- ☞ **Consequences:** Results, trade-offs of use
- ☞ **Implementation:** Pitfalls, hints, variations
- ☞ **Sample code**
- ☞ **Known uses**
- ☞ **Related patterns**

March 6, 2003

DIKU Project Seminar

Gang of Four patterns

- ☞ 23 design patterns
- ☞ Organized in 2 dimensions:
 - ☞ **Scope:**
 - ☞ class
 - ☞ object
 - ☞ **Purpose:**
 - ☞ creational (how to create objects)
 - ☞ structural (composition of classes and objects)
 - ☞ behavioral (ways of interaction and distribution of responsibility)

March 6, 2003

DIKU Project Seminar

General themes

- ⌘ **Loose coupling:** Add indirection to achieve decoupling, flexibility and reuse
- ⌘ **Abstraction:** Naming, encapsulation, parametrization (same purpose as above)
- ⌘ **Dynamic :** Delegation instead of implementation inheritance
- ⌘ **Object-orientation:** object-oriented (re)design
- ⌘ **Principles of object-oriented design:**
 - ⌘ “Program to an interface, not an implementation”
 - ⌘ “Favor object composition over class inheritance.”

March 6, 2003

DIKU Project Seminar

Reasons for redesign

- ⌘ Creating objects by specifying class explicitly
- ⌘ Dependence on specific operations
- ⌘ Dependence on hardware/software platform
- ⌘ Dependence on object representations and implementations
- ⌘ Algorithmic dependencies
- ⌘ Tight coupling
- ⌘ Extending functionality by subclassing
- ⌘ Inability to alter classes conveniently

March 6, 2003

DIKU Project Seminar

GoF patterns

- | | |
|--------------------|---------------------------|
| ⌘ Abstract factory | ⌘ Chain of Responsibility |
| ⌘ Builder | ⌘ Command |
| ⌘ Factory method | ⌘ Interpreter |
| ⌘ Prototype | ⌘ Iterator |
| ⌘ Singleton | ⌘ Mediator |
| ⌘ Adapter | ⌘ Memento |
| ⌘ Bridge | ⌘ Observer |
| ⌘ Composite | ⌘ State |
| ⌘ Decorator | ⌘ Strategy |
| ⌘ Facade | ⌘ Template Method |
| ⌘ Flyweight | ⌘ Visitor |
| ⌘ Proxy | |

March 6, 2003

DIKU Project Seminar

Structural and functional composition

- ⌘ **Composite:** Define tree-like structure, where internal and leaf nodes have same interface.
- ⌘ **Decorator:** Functional composition of objects with same interface.
- ⌘ **Chain of responsibility:** Chain objects and delegate requests along the chain.
- ⌘ **Proxy:** Define local object that delegates (most) functionality that other objects.

March 6, 2003

DIKU Project Seminar

From built-in methods to higher-order methods

- ⌘ **Command:** Turn requests into objects
- ⌘ **Interpreter:** Define (programming) language of requests
- ⌘ **Strategy:** Parameterize over different algorithms
- ⌘ **Visitor:** Receive and invoke any programmable operation on object (not just built-in methods)

March 6, 2003

DIKU Project Seminar

More design patterns...

- ⌘ **Singleton:** Make sure only one instance of a class exists during any program run (shared state)
- ⌘ **Adapter:** Connect two (existing) interfaces (have class with interface I1, and another one that needs I2)
- ⌘ **Flyweight:** Split object state into mutable (context) and immutable (flyweight) and share flyweights.
- ⌘ **Iterator:** Iterate over collection

March 6, 2003

DIKU Project Seminar

More design patterns...

- ✧ **Facade:** Centralize interface for collection of classes
- ✧ **Mediator:** Localize control, make centralized controller for orchestrating collaborating objects
- ✧ **Memento:** Checkpoint object (save state)
- ✧ **Observer:** Publish-subscribe interface for “pushing” state changes to arbitrary number of clients
- ✧ **State:** Object-oriented state representation for finite state machines

March 6, 2003

DIKU Project Seminar