Design Engineering

Overview

- What is software design?
- How to do it?
- Principles, concepts, and practices
- High-level design
- Low-level design

Design Engineering

- The process of making decisions about HOW to implement software solutions to meet requirements
- Encompasses the set of concepts, principles, and practices that lead to the development of high-quality systems

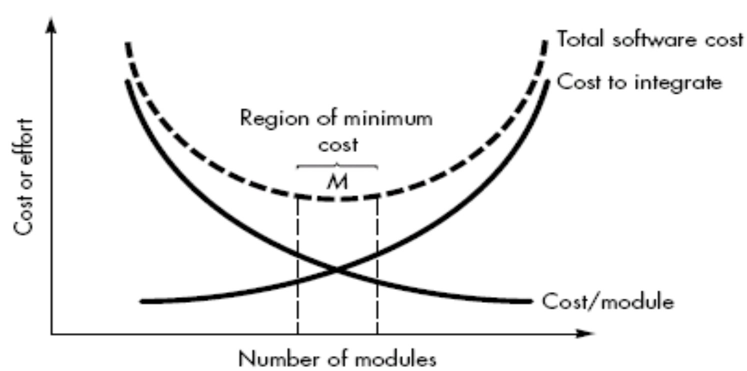
Concepts in Software Design

- Modularity
- Cohesion & Coupling
- Information Hiding
- Abstraction & Refinement
- Refactoring

Modularity

- Software is divided into separately named and addressable components, sometimes called modules, that are integrated to satisfy problem requirements
- Divide-and-conquer

Modularity and Software Cost



Cohesion & Coupling

- Cohesion
  - The degree to which the elements of a module belong together
  - A cohesive module performs a single task requiring little interaction with other modules
- Coupling
  - The degree of interdependence between modules
- High cohesion and low coupling

Information Hiding

- Do not expose internal information of a module unless necessary
  - E.g., private fields, getter & setter methods

Abstraction & Refinement

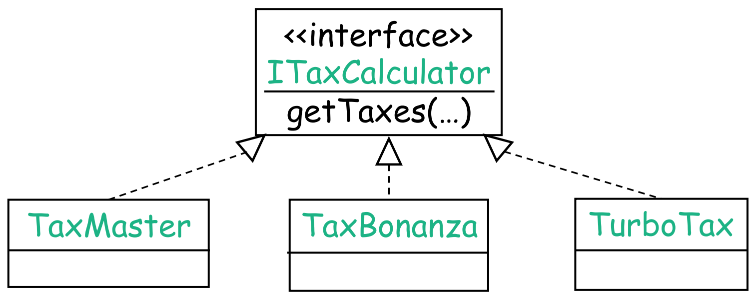
- Abstraction
  - To manage the complexity of software,
  - To anticipate detail variations and future changes
- Refinement
  - A top-down design strategy to reveal low-level details from high-level abstraction as design progresses

Abstraction to Reduce Complexity

- We abstract complexity at different levels
  - At the highest level, a solution is stated in broad terms, such as "process sale"
  - At any lower level, a more detailed description of the solution is provided, such as the internal algorithm of the function and data structure

Abstraction to Anticipate Changes

- Define interfaces to leave implementation details undecided
- Polymorphism



Refinement

- The process to reveal lower-level details
  - High-level architecture software design
  - Low-level software design
    - Classes & objects
    - Algorithms
    - Data
    - UI

Refactoring

"…the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure" --Martin Fowler

- Goal: to make software easier to integrate, test, and maintain.

Software Design Practices Include:

- Two stages
  - High-level: Architecture design
    - Define major components and their relationship
  - Low-level: Detailed design
    - Decide classes, interfaces, and implementation algorithms for each component

How to Do Software Design?

- Reuse or modify existing design models
  - High-level: Architectural styles
  - Low-level: Design patterns, Refactorings
- Iterative and evolutionary design
  - Package diagram
  - Detailed class diagram
  - Detailed sequence diagram

Software Architecture

- "The architecture of a system is comprehensive framework that describes its form and structure -- its components and how they fit together"

  --Jerrold Grochow

What is Architectural Design?

- Design overall shape & structure of system
  - the components
  - their externally visible properties
  - their relationships
- Goal: choose architecture to reduce risks in SW construction & meet requirements

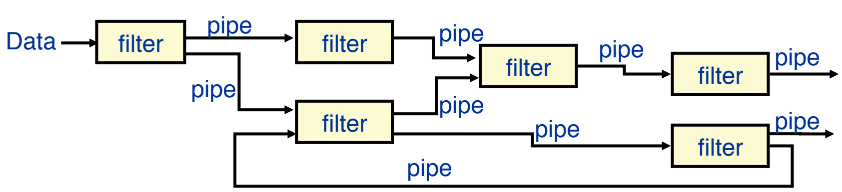
SW Architectural Styles

- Architecture composed of
  - Set of components
  - Set of connectors between them
    - Communication, co-ordination, co-operation
  - Constraints
    - How can components be integrated?
  - Semantic models
    - What are the overall properties based on understanding of individual component properties?

Architecture Patterns

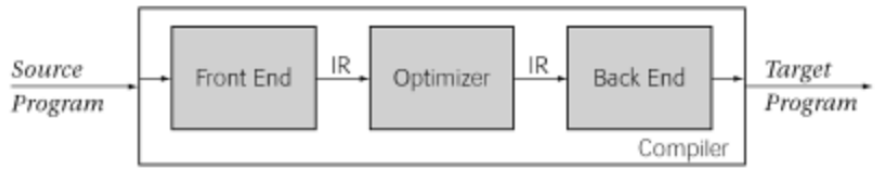
- Common program structures
  - Pipe & Filter Architecture
  - Event-based Architecture
  - Layered Architecture

Pipe & Filter Architecture

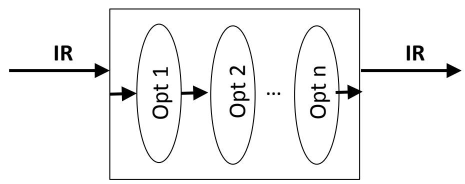


- A pipeline contains a chain of data processing elements
  - The output of each element is the input of the next element
  - Usually some amount of buffering is provided between consecutive elements

Example: Optimizing Compiler



Compiler Structure



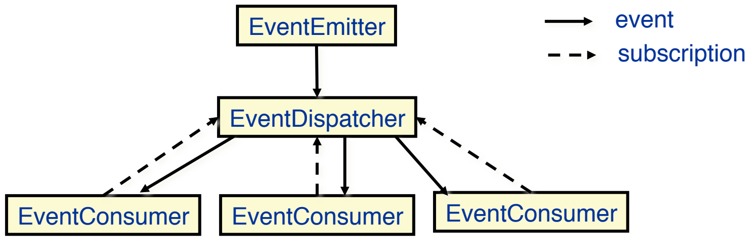
Compiler Optimization

[Engineering a Compiler, K. D. Cooper, L. Torczon]

Pros and Cons

- Other examples
  - UNIX pipes, signal processors
- Pros
  - Easy to add or remove filters
  - Filter pipelines perform multiple operations concurrently
- Cons
  - Hard to handle errors
  - May need encoding/decoding of input/output

Event-based Architecture



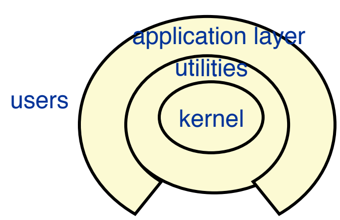
- Promotes the production, detection, consumption of, and reaction to events
- More like event-driven programming

Example: GUI

Pros and Cons

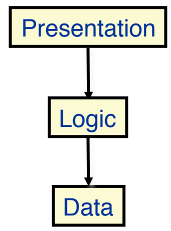
- Other examples:
  - Breakpoint debuggers, phone apps, robotics
- Pros
  - Anonymous handlers of events
  - Support reuse and evolution, new consumers easy to add
- Cons
  - Components have no control over order of execution

Layered/Tiered Architecture



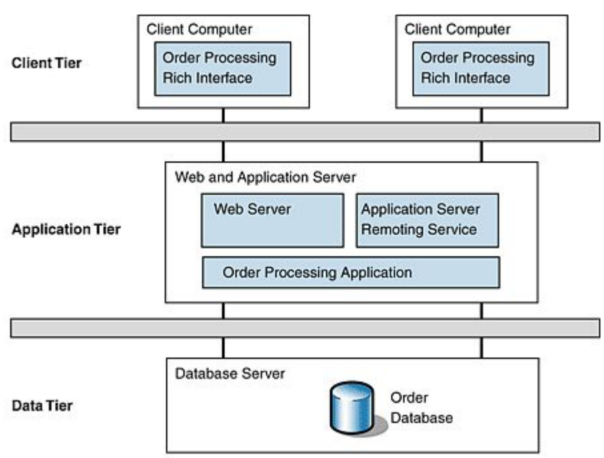
- Multiple layers are defined to allocate responsibilities of a software product
- The communication between layers is hierarchical
- Examples: OS, network protocols

3-layer Architecture



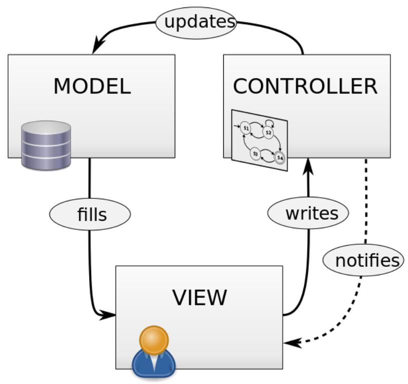
- Presentation: UI to interact with users
- Logic: coordinate applications and perform calculations
- Data: store and retrieve information as needed

Example: Online Ordering System



http://www.cardisoft.gr/frontend/article.php?aid=87&cid=96

Model-View-Controller



Design of Finite State Machine Drawing Tool

https://commons.wikimedia.org/wiki/File:MVC_Diagram_(Model-View-Controller).svg

Key Points about MVC

- View layer should not handle system events
- Controller layer has the application logic to handle events
- Model layer only respond to data operation

Layered Architecture: Pros and Cons

- Pros
  - Support increasing levels of abstraction during design
  - Support reuse and enhancement
- Cons
  - The performance may degrade
  - Hard to maintain

Detailed Design

- To decompose subsystems into modules
- Two approaches of decomposition
  - Procedural
    - system is decomposed into functional modules which accept input data and transform it to output data
    - achieves mostly procedural abstractions
  - Object-oriented
    - system is decomposed into a set of communicating objects
    - achieves both procedural + data abstractions

Work Results

- Dynamic models
  - help design the logic or behaviors of the code
  - UML interaction diagrams
    - (Detailed) sequence diagrams, or
    - Communication diagrams
- Static models
  - help design the definition of packages, class names, attributes, and method signatures
  - (Detailed) UML class diagrams

OOD

- To identify responsibilities and assign them to classes and objects
- Responsibilities for doing
  - E.g., create an object, perform calculations, invoke operations on other objects
- Responsibilities for knowing
  - E.g., attributes, data involved in calculations, parameters when invoking operations

Guidelines

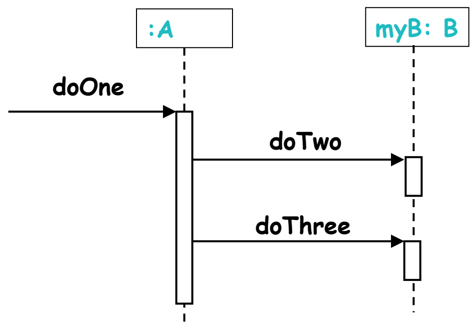
- Spend significant time doing interaction diagrams, not just class diagrams
- Do static modeling after dynamic modeling

UML Interaction Diagrams

- To illustrate how objects interact via messages
- Two types of interaction diagrams
  - Sequence diagrams
  - Communication diagrams

Sequence diagram

- Illustrate interactions in a kind of fence format, in which each new object is added to the right

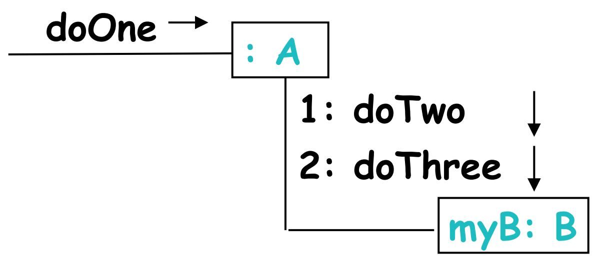


What Is The Possible Representation in Code?

```
public class A
{
    private B myB = new B();
    public void doOne()
    {
        myB.doTwo();
        myB.doThree();
    }
}
```

Communication Diagram

- To illustrate object interactions in a graph or network format, in which objects can be placed anywhere on the diagram



Sequence vs. Communication

- Sequence diagram
  - Tool support is better and more notation options are available
  - Easier to see the call flow sequence
- Communication diagram
  - More space-efficient
  - Modifying wall sketches is easier

Design Class Diagrams

- Differences from Conceptual Class Diagrams in Domain model
  - Contain types, directed associations with multiplicities, methods
  - Provide visibility between objects

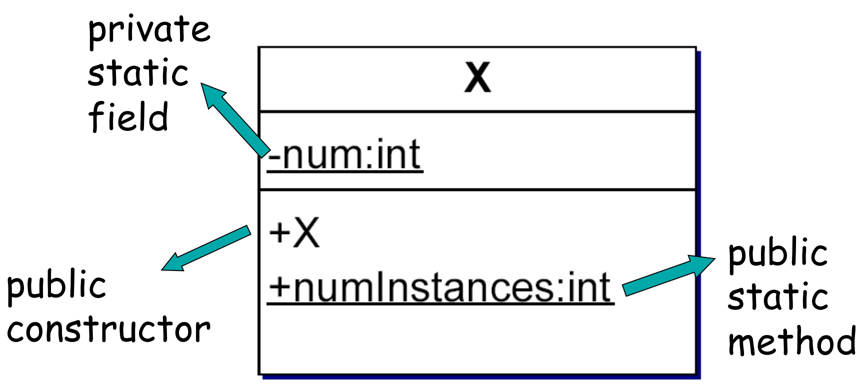
Type Information

- Types of attributes
- Types of method parameters/returns (can be omitted)

Accessibility of Methods and Fields

- public: can be accessed by any code
  - UML notation: +foo
- private: can be accessed only by code inside the class
  - UML notation: -foo
- protected: can be accessed only by code in the class and in its subclasses
  - UML notation: #foo
- Fields usually are not public, but have getters and setters instead

UML Class Diagram

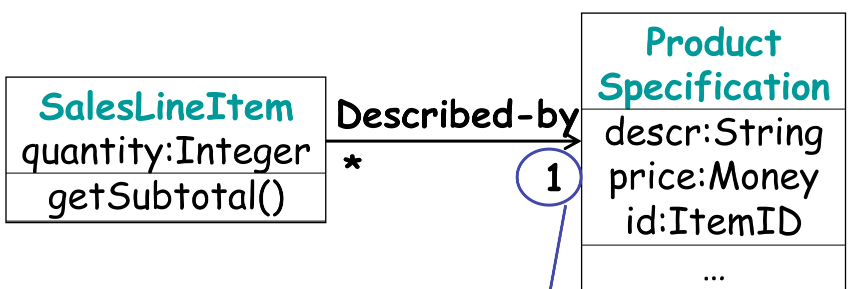


note: "static constructor"
is meaningless: by definition,
a constructor is invoked on an object

Mapping Design to Code

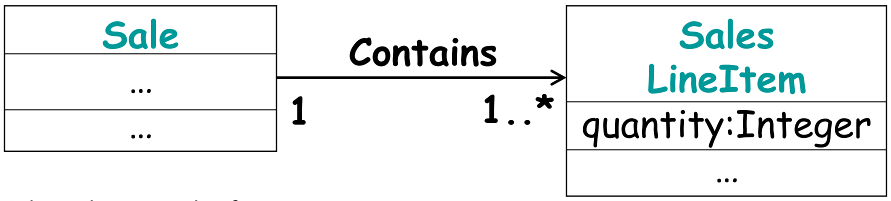
- DCDs -> classes in code
  - DCD: class names, methods, attributes, superclasses, associations, etc.
  - Tools can do this automatically
- Interaction diagrams -> method bodies
  - Interactions in the design model imply that certain method calls should be included in a method's body

Mapping Associations (* : 1, 1 : 1)



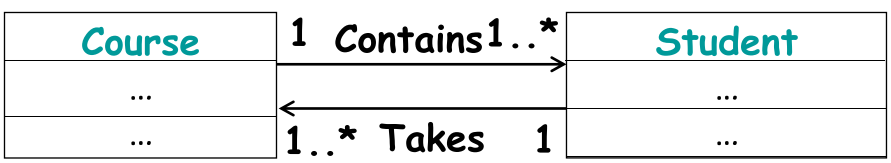

```
public class SalesLineItem {
   private int quantity;
   private ProductSpecification productSpec;
   public SalesLineItem(ProductSpecification s, int q) {…}
}
```

Mapping Associations (1 : *)




```
public class Sale {
    private List<SalesLineItem> lineItems = new
        ArrayList<SalesLineItem>();
    private Date date = new Date();
    public void makeLineItem(ProductDescription desc, int qnty) {
        lineItems.add(new SalesLineItem(desc, qnty));
    }
    …
}
```


Mapping Associations (* : *)




```
public class Course {
    private List<Student> students = new ArrayList<Student>();
    public addStudent(int sid) {…}
}

public class Student {
    private List<Course> courses = new ArrayList<Course>();
    public addCourse(int cid) {…}
}
```

Design Patterns

- Definition
  - A named general reusable solution to common design problems
  - Used in Java libraries
- Major source: GoF book 1995
  - "Design Patterns: Elements of Reusable Object-Oriented Software"
  - 24 design patterns

Purpose-based Pattern Classification

- Creational
  - About the process of object creation
- Structural
  - About composition of classes or objects
- Behavioral
  - About how classes or objects interact and distribute responsibility

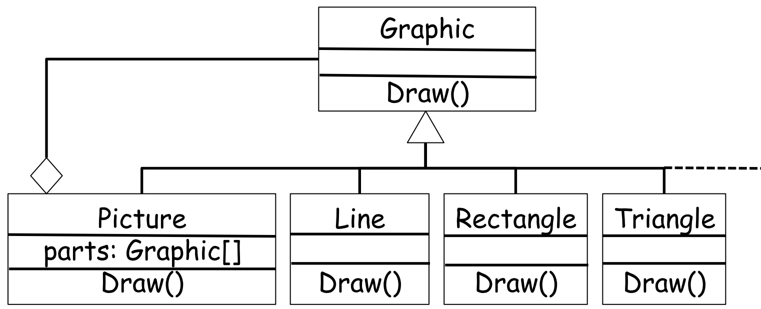
Design pattern space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Visitor Pattern

- Scenario: Given a set of objects in a heterogeneous aggregate structure, such as a tree, you want to define and perform various distinct and unrelated operations on them

Example



- What will you do if you always need to add operations to the objects, such as Add(), Remove(), Update()?

Visitor Pattern

- You want to limit the scope of introduced changes
  - Within a class vs. across different classes
- You want to avoid "polluting" the node classes with various operations
- Create a Visitor class hierarchy that defines a virtual visit() method for each node type
- Add a virtual accept() method to the base class of all node classes

Visitor Pattern