Handout 9

Object Oriented Design (OOD)

Fundamental Concepts of Software Systems Design

The Goal: produce a specification of an entity in sufficient detail to permit its physical realization.

Good design:
- minimizes the total cost of the system over its entire lifespan (50-90% of the cost will be incurred after the system is put into the operation).
- creates reusable components

Good designs achieve a balance of flexibility, generality, efficiency, optimality, portability, robustness, elegance, clarity, …

Design specification describes
Organization of the system into subsystems
- modular program structure
- control relationships between modules
Approach to managing the data stores
Human-computer interface

It also provides a specification for
Data Structures
Algorithms
used to implement the required functionality.

Design Principles

Abstraction levels and Refinement

Design is created in an iterative process: starting with the specification at a highest level of abstraction (conceptual), it is refined by focusing on the internal structure of information and procedural detail.

Modularity

Software is divided into separately named and addressable components, called modules. Modularity allows software to be intellectually manageable, reduces complexity and facilitates change. Monolithic (uni-module) software is difficult (costly) to build, test and maintain.
How to decompose the software to the best set of modules?

There is no “perfect decomposition”. Try to achieve Functional Independence of module, strive for high Cohesion and low Coupling between the modules.

**Cohesion**: a degree to which a single module performs a single function (task) within the system. A highly cohesive module “does one thing”. Cohesion is a measure of relative functional strength of a unit of software. Cannot be measured directly, but should direct the modular decomposition.

**Coupling**: a measure of the number and complexity of connections among modules in a software structure. High degree of coupling implies a lot of inter-module connections, thus

- errors that occur in one module are easily propagated to the connected modules
- changes in the interface of one module require changes in the other

**Information Hiding**

Modules should be specified in such a way that information (data and procedures) contained within a module are inaccessible to other modules that have no need for such information. *Interface* refers to the publicly available ways of using a module.

If an entity is hidden, then modifications to it will not require changes to the rest of the system. In other words, the fewer operations “know” about the entity, the less likely they are to be affected by modifications to the entity.

**Drawbacks in structured Analysis and Design**

- Need to reconcile Data Flow Diagram representation of processes with Entity-Relationship Diagram representation of data
- Resulting discontinuity between the processes of Analysis and Design.

**A few benefits from OO**:

- Uniform underlying representation means Design is a natural continuation of the Analysis process.
- Class&Objects create natural decomposition into highly cohesive modules.
- Encapsulation & Information Hiding promote functional independence of modules, decreased coupling
- Inheritance – enforces cohesion, makes the system easily extendable, helps localize maintenance (changes and testing)
- Polymorphism -- contributes to clarity of design, independence of modules, simplifies extension
OO Design

Coad-Yourdon:
Augment the OOA model with additional Classes&Objects to handle
implementation-related activities of
1. Human Interaction Component
2. Problem Domain Component
3. Task Management Component
4. Data Management Component

Problem Domain Component is largely the result of OOA, but may need modifications
under certain circumstances. Following changes are sometimes appropriate

- Use available design/programming classes (reuse)
  treat OOA classes as subclasses of the available library classes taking full
  advantage of attributes and services of preexisting classes
- Group domain-related classes together under an umbrella of a common
  superclass to provide a common protocol for communicating with Data
  Management Component or other external system.
- Re-structure the Class&Objects that are defined via multiple inheritance to
  accommodate implementation in a language without multiple inheritance.
  Replace multiple inheritance with delegation.
- Modify design to improve performance(message passing creates probably
  the most significant overhead):
  - “squash” two classes together to avoid overhead of message
    passing
  - violate principle of encapsulation by providing global data areas,
  - add redundant classes/attributes/services for efficiency to avoid re-
    calculation
  - …

but don’t create a performance crisis if there isn’t one.
It’s easier to make a working system efficient rather than it is to make
an efficient system to work.
80% of time-overhead is usually concentrated in 20% of code, so it is
useful to develop a prototype and use profiling to find out the
bottlenecks.