Designing Large-Scale Client/Server Systems with UML

Leszek A. Maciaszek
Macquarie University, Sydney, Australia

Topics
- What is design?
- C/S architectures
- Solution strategies
- System components
- Component cohesion & coupling
- MVC components
- User interface design
- Database design
- C/S program design
- Designing maintainable and scalable systems

What is design?
- Low-level model of system’s:
  - architecture (architectural design)
  - internal workings (detailed design)
- Architectural design:
  - deployment strategy (UML nodes)
  - modularization (UML packages)
- Detailed design
  - client (GUI)
  - server (database)

C/S concept
- Client - an application requesting data from another node
  - a node with an application process linked with the client part of the DBMS
  - it may or may not have a local database
- Server - any node with a database, of which data may be requested
  - A node with a database can be a client in some transactions and a server in others

C/S solution strategies
- Client
  - native database interface as provided by many 4GLs (e.g., PowerBuilder, Developer 2000, Delphi)
  - ODBC or JDBC driver to a database (e.g., Visual C++, Power J++)
- Server
  - relational database (e.g., Sybase, Oracle, informix)
  - object-relational database (e.g., Oracle8, Illustra, Ingres)
  - object database (e.g., ObjectStore, Versant, Objectivity/DB)

Two meanings of C/S
- In business applications (RDBMS)
  “Think remote procedures not remote data”
- in workgroup computing (ODBMS)
  Think server data on a client
Five C/S models

Three-tiers: pros and cons

Pros:
- Flexibility and scalability
- End-user independence
- Low upgrade cost
- Natural fit for enterprise-wide applications

Cons:
- Additional points of failure
- Performance bottlenecks
- Security exposures

Example of three-tier

MVC - Model, View, Controller

Controller package
Two relational tables in monitoring package

Using class diagram for object-relational DB design

Designing collaboration of client objects ...

... and server objects

Round-trip engineering with client program

Round-trip engineering with database
Example of reverse engineering session

Re-engineering to and round-trip engineering with ORDB

Theory and practice of inheritance

Inheritance supports re-use but at the same time it opens the proverbial “can of worms.”
Consider the following questions:
• Should the creator of a class be prevented from changing the definition of the class, once any other user derived a subclass from the class?
• Should the creator of a class be given access privileges to all direct and indirect subclasses of the class?
• What naming conventions need to be used and imposed on class developers so that to avoid future name clashes due to subclassing?

No dynamic and no multiple classification

Inheritance and databases

Inheritance conflicts with databases

Databases are designed to store and manage lots of objects belonging to relatively small number of classes
Inheritance prefers an exactly opposite situation - few objects, many classes
If generalization is used as the main modeling construct, the database intention grows quickly to the size that is unmanageable and unproportionally large to its extension

In most OO-implementation languages, objects cannot:
• change class (no dynamic classification)
• belong to more than one class (no multiple classification).
The repercussions are severe whenever objects persist longer than a single execution of a single program.

Inheritance supports re-use but at the same time it opens the proverbial “can of worms.”
Consider the following questions:
• Should the creator of a class be prevented from changing the definition of the class, once any other user derived a subclass from the class?
• Should the creator of a class be given access privileges to all direct and indirect subclasses of the class?
• What naming conventions need to be used and imposed on class developers so that to avoid future name clashes due to subclassing?

In most OO-implementation languages, objects cannot:
• change class (no dynamic classification)
• belong to more than one class (no multiple classification).
The repercussions are severe whenever objects persist longer than a single execution of a single program.

The repercussions are severe whenever objects persist longer than a single execution of a single program.
The semantics of aggregation is stronger and it...
Aggregation facilitates encapsulation

- The hierarchical composition in natural systems can be seen as having a large number of extensively interlinked objects, but few classes:
  - There are thousands or millions of objects of class ‘molecule’ that combine to form an object of class ‘organelle’.

- Aggregation can model large quantities of objects by grouping them and establishing part-whole relationships between them and placing them in adjacent stratified layers.

- In this architecture, the higher aggregation layers hide (encapsulate) the complexity of object interactions in lower layers.

References