CSC 520 - Advanced Object Oriented Programming, Spring 2018

Subsystem Design
Subsystem Design

- Subsystem design is unique to each project.
- Here we will consider design of the business layer:
  - decide what objects will populate the layer,
  - how the objects will be connected,
  - what the object interfaces will be.
- The goal is to transform the business-oriented class model into an implementation-oriented classes.
Subsystem Design Activities

- Design the classes and fields of the business layer, using the analysis class model as a guide.
- Decide how any persistent data will be stored and design the storage layout.
- Finalize the look and feel of the user interface.
- Walk through the system use cases with respect to the UI design and note any business services that must be supported by the middle tier.
- Develop the business services into server objects whose messages are available over the network.
- Consider the measures that are needed to ensure concurrency control.
Mapping the Analysis Model to the Design Model Activities

- Mapping operations
- Variable Types
- Visibility of Fields
- Accessors
- Mapping Classes, Attributes and Compositions
- Mapping Other Types of Association
The analysis operations should be ignored for the purpose of design because the operations were introduced as a side effect of verifying that analysis classes would support implementation.

Messages should be added for these reasons:
- To allow client objects to read or change the values of fields.
- To allow client objects to access derived data.
- Because our experience or intuition tell us that a particular message might be useful.
- Because some framework or pattern that we have decided to use requires certain messages to be present.
During design, we have to decide what type each field should have.

This can usually be restricted to the following types:

- The primitive types and simple classes that we would expect to find in every object-oriented programming language.
- Classes that we are designing.
- Classes from the patterns and frameworks that we have chosen to use.

Note: in some languages arrays and collections do not mix well.
The visibility of a field specifies which pieces of code are allowed to read or modify the value.

**Visibilities:**
- **Private** (- in UML): only visible in the defining class.
- **Package** (˜ in UML): visible within the defining class and to all classes in the same package.
- **Protected** (# in UML): visible within the defining class, to all classes in the same package, and to all subclasses of the defining class.
- **Public** (+ in UML): visible everywhere.
Visibility of Messages

- **Private**: if its implementation code is for use by this class only
- **Package**: if its implementation code is to be used by the class itself and by classes in the same package
- **Protected**: if its implementation code is to be used by the class itself, by its subclasses, and by classes in the same package
- **Public**: if it is part of the interface of the package
■ It is good practice to provide accessor messages for fields.
■ There are two varieties of accessor messages:
  ■ getter: returns the value of a field
  ■ setter: set the field to a new value
■ Accessors allow us to centralize access to fields
Mapping Classes, Attributes and Compositions

- The same UML notation is used for analysis and design class diagrams, however design class diagrams typically use more of the available notation.
  - Classes from analysis may be removed during design
  - The attributes are given types
  - New supporting classes may be added

- The composition relationship is similar to an attribute: the owning object uses the services of the attribute or composed object.
Mapping Associations

- For mapping purposes, we do not need to make a distinction between aggregation and association because object-oriented programming languages make no distinction.
- Associations are implemented as attributes object-oriented programming languages; we need to decide the direction of the relationship.
- For a two-way relationship, we need to ensure that the each object is updated properly.
Mapping One-to-One Associations
Mapping One-to-Many Associations

possible mappings

1..* -a:List 1

1

mapping options:

1..* 1

-b

A  B  A  B

A  B

A  B
Mapping Many-to-Many Associations

possible mappings

- A
  - 1..*
  - -bs:List

- B
  - 0..*
  - -as:List

- A
  - 1..*
  - -as:List

- B
  - 0..*
  - -bs:List
Mapping Association Classes

- Association classes do not map to most object-oriented programming languages.
- Options for mapping association classes
  - Create a design class with two extra fields to point at the associated objects.
  - If we need to navigate from a class to an association class, we can add a field to the respective class (with the associated synchronization problems).
  - Add a message to a *home* class for the association class objects to retrieve the object at either end of the relationship (a home should be a singleton).
- An association class is the most general way of mapping *any* association from analysis to design.
Mapping Association Classes

Possible mapping:

A 1 C 1 B
Universal Identifiers

- Most business objects, at some point in their life, need to be retrieved by key.
- A key is an attribute value, or combination of values, that is unique to each instance.
- Handling keys can get cumbersome, so it is sometimes useful to introduce a unique number to distinguish each business object from other objects of the same class.
- These *universal identifiers* help synchronize copies of objects, track their movements as they travel around a network, and handle keys uniformly and efficiently.
Handling Persistence

- Most systems have data storage requirements; data must *persist* when the application is shut down.
- In object-oriented programming, we need to consider how graphs of run-time objects can be mapped to storable data.
- There is typically a semantic gap between the programming language that we use to write the code and the way we access the database.
- Once the business layer and data layer have been designed, we can consider the code needed to translate from one layer to the other.
A database is a collection of related data organized in a way that supports easily processing the data.

A Database Management System (DBMS) is software that provides the ability to create and manipulate a database.

Types of DBMS:
- indexed file
- hierarchical
- network
- relational
- object-oriented
Database Languages

- Data definition language: defines the data types and relationships among them
- Data manipulation language: performs operations such as inserting, updating, or deleting data.
- Query language: performs information retrieval
A relational database models data as relations, which are based on rigorous mathematics.

- A relation is organized as a table of columns and rows.
- Each column is also called an attribute.
- Each row is also called a record.
- Each row includes a unique key to identify that row.
The Structured Query Language (SQL) combines the roles of all database languages into a single language for relational databases.

- SQL is a *declarative* programming language.
- A declarative programming language describes *what* computation should be performed not *how* to compute it.
Mapping an Object Model to a Relational Model

- The following elements need to be considered when converting an object model to tables:
  - Mapping entity classes
  - Mapping associations
    - One-to-one
    - One-to-many
    - Many-to-many
    - Association classes
  - Mapping object state
  - Mapping inheritance

- The analysis model is closer to the relational model (associations are two-way), but the design model has types assigned to fields.
An entity (business object) can be mapped to the relational model by introducing a table with the same name as the entity’s class.

Each row of the table represents unique object from the business domain.

Each simple field can be mapped to a column in the table with the same name.

Introducing an integer attribute as the primary key simplifies the mapping and provides the benefits associated with universal identifiers.
One-to-One and One-to-Many Associations

- A one-to-one association can be mapped to the relational model by adding a foreign key to one of the entity tables.
- A foreign key is an entity in one table that refers to the primary key in another table.
- Alternatively, the two tables can be combined into one.
- Optional associations (multiplicities of 1 and 0..1) can be mapped by adding a foreign key to the optional end (databases support nullable columns).
- A one-to-many association can be mapped to the relational model by adding a foreign key to the ‘many’ table.
Many-to-Many Associations and Association Classes

- A many-to-many can be mapped to the relational model by introducing a link table.
- A *link table* represents a link between an entity in one table and an entity in another table.
- A basic link table has two columns: one for each entity’s foreign key.
- An association class can be mapped to the relation model by adding a link table with additional attribute columns.
Business objects that have an associated state machine need to have the state the object currently has recorded in the table.

The simple way is to include a state column with an integer or string type and represent each state accordingly.

Alternatively, a new table can be introduced for each state and foreign keys can be used to indicate the state of the object.

Some states have state attributes – data associated with a business object when it is in a particular state.

- state column: add a (nullable) column for every state attribute
- state as table: store state variables as extra columns
An inheritance hierarchy can be mapped to the relational model by introducing a table for each class with columns for the attributes added by that class.

The tables must share the same primary key to ensure that all the attributes for an object can be found.

The universal identifier is used for the primary key.

Alternatively, a single table can be used that has a nullable column for each attribute in the hierarchy.
Finalizing User Interfaces

- During requirements capture and analysis we may have produced:
  - user interface sketches
  - boundary objects in communication diagrams

- Most of a software system can be designed successfully without considering the user interfaces in detail because:
  - the correct behavior of the system depends on its internal construction, not on the way people interact with it,
  - the preference to produce reusable code.

- During design, we need to take coarse boundary objects and precise use cases and transform them into user interface descriptions that can be implemented directly.
Designing the Business Services

- Business services are the queries and commands that a middle tier makes available to its clients.
- The business services represent a concise summary of the information flow between the user interfaces on the client and the business logic on the middle tier.
- Business services can be derived by walking through the use cases, with respect to the user interfaces and the system architecture.
Client-Server Communication for Business Services

- Simplifies the client code; most client interfaces only need a subset of the system’s capability.
- Complicates the business layer; the business layer can be arbitrarily complex without impacting the client.
- Enables the construction of a pluggable server layer; we may want to provide various kinds of interfaces without reimplementing the middle tier services every time.
Proxies and Copies

- A business service may send or receive business objects.
- Logically, business objects flow back and forth between the client and server.
- There are two strategies for handling this:
  - proxy: client objects that know how to forward messages they receive to a real business object
  - copy: client objects containing a copy of the real business object’s data.
Proxies

- Advantages
  - All clients see the same objects
  - All run-time systems merge into a single universe; distributed objects are treated in the same way as local objects.

- Disadvantages
  - The business layer objects must be secured.
  - Network traffic increases.
  - The burden on the middle tier increases.
Copies

■ Advantages
  ■ Network traffic decreases.
  ■ Processing does not take place in the middle tier.
  ■ Direct access to business methods does not have to be secured.
  ■ The object’s implementation does not need to be concurrent safe.

■ Disadvantages
  ■ Too much data may be copied.
  ■ The copies can diverge because each client has an independent copy of every object that it uses.
Business Services

- Lightweight copies:
  - when a client needs to identify a business object to the server, it passes the universal identifier of the object.
  - a response from the server only contains essential information.

- Session identifiers can be used for a level of security when a user logs on to a system.

- Business services are realized using sequence diagrams.