cs2704
Object-Oriented Software
Design and Construction

Ray A. Reaux
rreaux@vt.edu
629 McBryde
Object-Oriented Strategies

- The strength of object oriented analysis, design, and programming is the ability to manage software complexity:
  - in conception,
  - in development, and
  - in operation.

- Four OO Strategies:
  - Abstraction -- simplifying to essentials.
  - Separation -- separating “what” from how.
  - Composition -- assembling simple components into a more complex object.
  - Generalization/specialization -- identifying common properties among objects.

• Software is inherently complex.
• Strategies are needed to reduce and/or control this complexity.
• Previous attempts, particularly in procedural languages include design strategies such as top-down or bottom-up design and programming techniques such as procedures and separately compiled modules.
• Object-oriented builds on and extends these.
Strategies, Structures, and Goals

- Replaceability -- implementation can be separated from the design, allowing transportability and updates.
- Extensibility -- system can easily grow to encompass new functions.
- Reusability -- design and software can be recycled, thus reducing cost.
- Flexibility -- design and software can be easily changed without having adverse impact on the rest of the system.
Abstraction

“A named, tangible representation of the attributes and behavior relevant to modeling a given entity for some particular purpose.”

“In object-oriented analysis and design, abstraction provides a mechanism for managing and communicating complexity.”

“In object-oriented programming, abstraction provides a mechanism for definition reuse and encapsulation.”

• An abstraction:
  • is tangible, not ethereal. Forms concrete manifestations of abstract concepts.
  • focuses on essential aspects of an entity and ignores or conceals less essential properties.
  • accurately models the real world.

• Example: Abstracted web server:
  • accepts requests for a web page.
  • sends requested web page to client requesting it.
  • allow multiple connection.

• Less essential elements of a web server:
  • listens on ports 3232.
  • supports only 8 simultaneous “connections”.
**Abstraction (2)**

- A single entity may have many abstractions and choosing which attributes and behaviors to include in the abstraction is critical.
- High level design scenarios:
  “The web server listens in on a specific port. When it receives a request for a web page, it spawns a PageFetcher thread which gets the page from the public_html directory and sends a copy of the page to the client requesting the page.”
- Booch’s Object Model:

  ![Object Model Diagram]

- The software engineer’s job is to identify valid abstractions and choose appropriate ones to use.
- Abstractions:
  - in early design stage, hand-made drawings, diagrams, and lists can be used.
  - in implementation, code is the abstraction.
  - software is abstracted into attributes and behavior.
- A definitive OOA and OOD reference:
  - Now part of the Unified Modeling Language (UML), the best known OOD model used in the US
• Attributes of an abstraction are mapped to a set of data called fields (variables, array, lists, complex data structures, etc.)

• Behavior of an abstraction is mapped to a set of methods (also known as operations, functions, actions).
Properties of a Good Abstraction

- **Well named** -- the meanings, intuitions, impressions, and expectations implied by a name accurately reflect the use of the abstracted entity.
- **Coherent** -- the abstraction contains a related set of attributes and behavior that make sense from the viewpoint of the user.
- **Accurate** -- the abstraction contains only those attributes or behavior that are part of the entity being modeled.
- **Minimal** -- the abstraction does not contain extraneous (unnecessary) attributes or behavior (unless planning for the future).
- **Complete** -- the abstraction contains all of the attributes and behavior necessary for it to perform its intended purpose.

- Well-named: don’t call a web server “squid” or “apache.”
- Coherent: don’t include in a web server a telnet server. Instead, separate the telnet server into another abstraction which interacts with the web server.
- Accurate: don’t stick into a web server a utility to rename files.
- Minimal: don’t put into a server a utility for doing cron jobs.
- Complete: make sure that a web server doesn’t have to make use of an FTP to send requested page.
Separation

“Distinguishing between a goal or effect and the means or mechanism by which the goal or effect is achieved.”

“In object-oriented analysis and design, separation allows management of complexity and parallel development.”

“In object-oriented programming, separation is the independent specification of an interface and one or more implementations of that interface. Separation provides for plug-and-play.”

• In design, allows deferment of implementation details to get the broader picture.
• Separation allows for parallel development of design and code.
• It enables easy upgrade of algorithms (implementations).
• Example:
  • providing interface definitions for a class and building to those interface specification.
  • in effect, software libraries
Separation (2)

- Separating "what" is to be done from "how" it is to be done.
- A goal or effect is often simpler to explain than the means needed to achieve it.
- Separation forces much of the complexity of software development into integration.
- Separation provides:
  - interchangeable implementations;
  - portability and easy upgrade;
  - simultaneous development and division of labor.
  - design and software reuse.

• Separation is the strategic justification for encapsulation.

• Because of separation, one object can use another object if it knows its name (associates with the other object) and knows its interface (the externally visible façade of the object).
Properties of a Good Separation

- **Well matched** -- the inputs, effects, and outputs of an implementation accurately reflect the interface specifications.
- **Predictable** -- separation should produce consistent results across implementations.
- **Complete** -- an interface specification transmits all information necessary for an implementation to work across its boundary.
- **Parsimonious** -- the information that has to cross the interface should be minimal.

- Well-matched: an algorithm implemented for a sort function should sort items.
- Predictable: no unexpected side-effects.
- Complete: parameters of a method should transmit all necessary information for a function to do its work.
- Because complexity of software is shifted to integration of objects, the minimal amount of information required to transmit across the boundaries of the separation, the easier it is to integrate the objects.
Composition

“Composition is the buildup of a more complex object from smaller, less complex objects.”

In object-oriented analysis and design, composition provides a mechanism for managing complexity.

“In object-oriented programming, it provides a mechanism for component reuse.”

• Composition allows for:
  • component reuse,
  • plug-and-play,
  • updating by component swapping.

• Example:
  • a web server is composed of a port listener, a socket connector, a HTML file reader, a HTML file sender.
  • an office-automated LAN is composed of workstations, printers, modems, file servers, etc.
• Associations mean that one object knows about (has some reference) to another object and therefore can communicate with that object.

• Aggregation means that an object is instantiated or exists within the context of another container object. Aggregation limits the scope of the contained object.

• From outside, an aggregated object is opaque; objects inside cannot be directly accessed.

• Associated objects may often be involved in several associations. It is a logical grouping, not a physical grouping like an aggregation.
Generalization/Specialization

“Generalization/Specialization is the widening/ narrowing of definitions that allow reuse of definitions.”

“In object-oriented design, it provides a mechanism for top-down/bottom-up analysis.”

“In object-oriented programming, generalization/ specialization is the basis for inheritance and polymorphism.”

• Generalization relies on recognizing patterns
• Generalization allows the software engineer to “zoom-out” in specification
• Specialization is the opposite, allowing the software engineer to “zoom-in” in specification
• Generalization/Specialization naturally leads to hierarchies, and this hierarchy is used as the basis for inheritance.
• In software, specialization usually means that an object definition lower in the hierarchy has more attributes than an object higher in the hierarchy.
Generalization (2)

- Generalization:
  - logical classification based on common characteristics
  - provides reuse of definition:
    - classes as templates for objects
    - inheritance of classes
- Specialization should provide additional capabilities to that of its generalized model.
- mechanisms for generalization
  - inheritance -- structuring of abstractions.
  - genericity -- parameterized classes (templates in C++)
  - patterns -- grouping of objects/classes with similar behavior

- Generalization allows the software engineer to “zoom-out” in specification
- Specialization is the opposite, allowing the software engineer to “zoom-in” in specification
- Generalization/Specialization naturally leads to hierarchies.
Characteristics of an Object

- In the real world, an object is anything we can touch, sense, or comprehend intellectually.
- In software, candidates for objects can be an idea or created construct that performs some specialized purpose.
- All objects should have the following characteristics.
  - an object has a unique identity;
  - an object has definable states;
  - an object exhibits some well-defined behavior;
  - an object has scope, that is, it occupies space and time.

- In the real world, an object is anything we can touch, sense, or apprehend intellectually.
- An object in software can be an idea or construct that performs some specialized purpose.
- Concepts such as time, beauty, or color are not objects. They are characteristics that describe an object.
- Contrast the three mentioned concepts with a clock, a sunset, and a red apple which are objects that have or make use of the three characteristics in some way.
Object Identity

- Identity provides the following:
  - bounds the object and gives it form,
  - differentiates the object from others,
  - provides some basis for object classification.

- Object identity is crucial if objects are to communicate with each other.
  - an object can have many references “pointing” to it, but a reference cannot point to many objects at the same time.

Identity is that property of an object that distinguishes it from another object.

• An object’s identity is specified by its object reference.

• An object reference is a variable of the same type as the object which “points” to an object of that type.

• In order for an object to exist, it must have at least one object reference “point” to it.

• An object may have several names (object references) within the “name space”, but each name (object reference) in the “name space” can only identify one object.
Object State

- An object is influenced by its history.
- It is state driven, meaning that the order in which one operates upon the object is important, and that the object transitions from one state to another depending on inputs made to the object.
- The states of an object are represented by attributes which have some value. These attributes may be:
  - static (constant persists with one value throughout the existence of the object) or
  - dynamic (they change based on some input to the object).

*The state of an object encompasses all static properties of the object plus the current dynamic values of each of these properties.*

- States are represented in code by current values of object attributes or fields.
- Transition from one state to another is dictated by the behavior (implemented in its methods) that the object experiences.
- This simplified assumption of transition is dependent on the object changing its own state, not having its attributes changed directly by some external object (i.e., no public attributes or fields).
Object Behavior

- To characterize an object’s behavior, we have to represent its outwardly visible and testable activity.
- A client object communicates with the server object by message passing.
- The message that is passed (a method invocation) is represented by:
  - the name of the object that is the server,
  - the method name, and
  - the set of parameters.

Behavior is how an object acts and reacts, in terms of state changes and message passing.

- No object exists in isolation.
- Objects are acted upon, and themselves act upon other objects.
- The operations that other objects may request an object to perform are generally called methods.
- An object’s available behavior is a subset of its total behavior and is determined by its current state.
- Objects follow the client server model.
  - A client asks another object to perform a service.
  - A server performs the service and returns the results.
Object Scope

- Because an object has identity, it takes up:
  - space, not only in actual memory but also in the name space;
  - exists for a limited lifetime and within a specified context, such as encapsulation within other objects or code blocks
- Scope determines:
  - limits on what other objects can interact with an object;
  - the mechanism for recapture of scarce resources.

*Scope is the lifetime or persistence of an object.*

- In Java, an object exists until it is no longer referenced (i.e., goes out of scope).
- Java implements Garbage collection so you don’t have to deallocate (delete) objects. This eliminates problems of
  - pointing to objects that don’t exist
  - unused objects lying around (memory leak).
- In contrast, in procedural languages:
  - an object’s scope is determined primarily by nesting within procedures.
  - explicitly allocated memory have to be deallocated.
Objects

- Object -- a distinct instance of a class which is structurally identical to all other instances of that class.
- An object is instantiated from a class.
- Object has:
  - implementation: the data and the implementation of the methods are hidden inside the object.
  - interface: the signature of all methods visible from outside the object. The sum of all public methods.

- In Java, an object is created with the `new` command and attached to an object reference:

```java
webServer WebServer1 = new webServer();
```

- An object reference is like a pointer that “refers” to an object of a specific class.
- Multiple object references can refer to the same object.
- A parameter passed to methods are actually copies of object references (unless basic data constructs).
• In most cases, an object has a copy of attributes defined by its class but share implementation of methods with other members of the class.

• Encapsulation -- compartmentalization of an abstraction to separate the interface from implementation.

• Encapsulation is more than information hiding. An actual barrier exists.

• Encapsulate as much as possible since it helps in portability, reusability, and system robustness (reduce side effects).
Data

- Data in an object can be of two forms:
  - fields (also called object variables),
  - internal method variables that have the scope of the method in which they are embedded.
- Fields have access protection (private, public, protected, package) which controls the encapsulation around it.
- Data field is uniquely identified within a name space.

- “Object level” data is called a field in Java.
- Every instance of an object has a copy of this field variable.
- A field value can be:
  - a primitive data type (integer, float, boolean).
  - an object reference (points to another object).

```java
private Port   PortListen;
private Socket  SocketConnected[];
public static final  int  MAX_CONNECTIONS = 8;
```

<table>
<thead>
<tr>
<th>Access Modifiers</th>
<th>Field type</th>
<th>Variable name or name of object reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Port</td>
<td>PortListen;</td>
</tr>
<tr>
<td>private</td>
<td>Socket</td>
<td>SocketConnected[];</td>
</tr>
<tr>
<td>public</td>
<td>static</td>
<td>final int MAX_CONNECTIONS = 8;</td>
</tr>
</tbody>
</table>
Methods

- Externally viewable collection of methods is called interface.
- Methods have access protection (private, public, protected, package).
- A method has a return value which can be a reference to an object, a primitive data type, or “void”.
- Methods can be overloaded -- a method name can be used to designate more than one method.
- A Java method is uniquely identified by its signature, a combination of:
  - name within the name space and
  - parameter set (number and type).

- A method returns an object reference or “void” (a special “null” object reference).
- Members without access modifiers default to package (accessible only to code and inherited by subclasses in the same package).
- Java method calls cannot have ellipsed or left out parameters (unlike C or C++).
- Methods cannot only differ by return type.

Valid overloaded methods:

```java
private OpCode ServePage(String PageName);
private OpCode ServePage(OpCode ResultCode);
public OpCode ServePage(String PageName, PageType Type);
```
A "class" represents all members of a given group.
Template from which objects are created.
A user defined “type”.
A type is a precise characterization of structural or behavioral properties which a group of entities share [Deutsch].
A class has:
- public parts -- reachable from the outside world.
- private parts -- unreachable from the outside world.

- In Java, the keyword “class” is used to define a new class.

- Structurally, each class has:
  - distinctive, meaningful name.
  - set of 0 or more fields (data items).
  - set of methods (operations).

class name_of_class {
  // comments explaining what this class is

  /** the body of the class definition goes in here between the curly braces **/
  /* set of 0 or more fields go here */
  /* set of 1 or more methods go here */

}
• Class -- named software representation that separates implementations of the representation from the interface of the representation.

• Interface -- a named software representation for an abstraction's behavior (black-box) intended to be implemented by one or more classes.

• Java uses the keyword “interface” for an abstraction of a class, with only declarations of its methods. Complete definition can be deferred.

```java
interface servePage {
    public CheckTreadCount();
    public void ServePage(String PageName);
    public void RequestPage(String PagenName);
}
```
Classes -- Examples

- As a general rule, you should make data private.
- An object’s data should be accessed (requested and returned) through a public method.
- Methods can call (invoke) other methods of the same entity.

```java
private void ServePage(String PageName) {
    PortListener Listener1 = new PortListener()
    Listener1.StartListen(PortNum Port);
    this.CheckThreadCount();
}
```