CPSC 5011: Object-Oriented Concepts

Lecture 1: Type, Abstraction, Information Hiding, Coupling, Cohesion, Class Design

Definitions

- Data type (or simply type) is a classification of data which tells the compiler or interpreter how the programmer intends to use the data
- Abstraction is the purposeful suppression, or hiding, of some details of a process or artifact, in order to bring out more clearly other aspects, details, or structure
- Information hiding is the purposeful omission of details in the development of an abstract representation. Information hiding is what allows abstraction to control complexity
- Coupling refers to the extent to which one component uses another to perform actions; generally a goal is to reduce coupling between software components
- Cohesion refers to the extent to which the actions of a component seem to be tied together in purpose; generally a goal is to increase cohesion within a software component

Classes of Data Types

- Primitive
  - Built-in to language implementation, i.e. numeric, boolean
- Composite
  - Derived from more than one primitive, combined as data structures, i.e. array, record (or tuple/struct), union, set, object
- Other
  - Enumeration
  - String and text types
  - Pointers and references
  - Function types
- Abstract data types (ADT)
  - Any type that does not specify an implementation, i.e. queue, set, stack, tree, graph, list, hash/map, etc. (an *interface)*

Other Considerations

- Type checking
  - Verify that operations performed on a type are meaningful and legal
  - Statically typed language, i.e. C++, Java, C#, Eiffel, Ada
    - Type is associated with identifier (i.e. `int x`)
    - Compiler verifies (no run-time overhead)
  - Dynamically typed language, i.e. Smalltalk, Python, Go
    - Type is associated with value (i.e. `value = 10; value = "hello";`)
    - Type tag checked at run-time
- Type casting
  - Implicit type casting (i.e. coercion)
  - Explicit type casting

Types in Software Design

- Procedural (structured) programming
  - Support clear control flow, functional decomposition, and ability to clearly define composite types (alternative to spaghetti code)
  - Little persistent connection between functions and data, using global data or passing parameters as the primary means to share data
- Modular programming
  - A module is a set of procedures with its related data
  - Promotes low coupling and high cohesion
    - Lowering the interdependency of two different software elements makes it more likely that a change in one element does not force a change in the second => this results in better software maintainability
    - Since change for a feature is restricted to a single module, cascading changes are avoided
    - For defined types, a separation of concerns (interface and implementation) reduces maintenance costs if modules were highly cohesive

Historical comparison

```
LIST

0 GOSUB 1000
10 LN=3000:GOSUB1100
20 LN=12345:GOSUB1100
999 END
1000 REM FIND JUMP TABLE GOSUB
1005 X=141:Y=44:L=2048:H=40959:W=1
1010 FORI=LTOH:IFPEEK(I)<>XORPEEK(I+W)<>
YTHENNEXTI
1020 JA=I+1:PRINT "PATCH ADDRESS IS";JA
1099 RETURN
1100 REM GOSUB TO LN
1110 LN$=STR$(LN)
1115 FORI=0TO5:POKEJA+I,32:NEXTI:PRINT"L
N$ IS";LN$
1120 FORI=0TOLEN(LN$)-1:POKEJA+I,ASC(RIG
HT$(LEFT$(LN$,I+1),1)):NEXT
1130 GOSUB,00000
1140 POKEJA,44:REM PUT COMMA BACK
1150 RETURN
3000 PRINT "LINE 3000":RETURN
12345 PRINT "LINE 12345":RETURN
READY.
```

Layers of Abstraction

- Levels of abstraction
  - Packages and name spaces
  - Clients and servers
  - Description of services
  - Interfaces
  - An implementation of an interface
  - A method in Isolation
- Finding the right level of abstraction

[See: Budd Ch 2, slides 5-12]

Additional Forms of Abstraction (1)

- Division into parts (has-a)
  - Takes a complex system, and divides it into component parts, which can then be considered in isolation
  - Examples:
    - A car has-a engine, and has-a transmission
    - A bicycle has-a wheel
- Specialization (is-a)
  - Takes a complex system, and views it as an instance of a more general abstraction
  - Examples:
    - A car is-a wheeled vehicle, which is-a means of transportation
    - A bicycle is-a wheeled
    - A pack horse is-a means of transportation

[See: Budd Ch 2, slides 13-16]

Additional Forms of Abstraction (2)

- Composition
  - A form of has-a; characterized by the following
    - Primitive forms
    - Rules for combining old values to create new values
    - The idea that new values can also be subject to further combination
  - Examples:
    - regular expressions, type systems, Windows, etc.
- Containers (holds-a)
  - Demonstrates that there is no *structural* or *semantic* relationship between the container and the contained thing
  - Examples of container types:
    - Array, List<T>, HashMap<String,Person>

[See: Budd Ch 2, slides 19-20]

Additional Forms of Abstraction (3)

- Patterns
  - Another attempt to document and reuse abstractions
  - Description of proven and useful relationships between objects; which can help guide the solution of new problems

[See: Budd Ch 2, slides 19-20]

Abstraction – Issues & Limitations

- Structures and procedures, i.e. `Stack`
  - To review, support clear control flow, functional decomposition, and ability to clearly define composite types
  - Libraries of procedures and functions provided the first hints of information hiding. They permit the programmer to think about operations in high level terms, concentrating on what is being done, not how it is being performed. But they are not an entirely effective mechanism of information hiding.
- Modules, i.e. two or more `Stack`s
  - Modules basically provide collections of procedures and data with import and export statements
  - Solves the problem of encapsulation -- but what if your programming task requires two or more stacks?

[See: Budd Ch 2, slides 23-25]

5 Criteria of Modularity

1. A software construction method satisfies Modular **Decomposability** if it helps in the task of decomposing a software problem into a small number of less complex subproblems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them
2. A method satisfies Modular **Composability** if it favors the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed.
3. A method favors Modular **Understandability** if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others.
4. A method satisfies Modular **Continuity** if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.
5. A method satisfies Modular **Protection** if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules.

[See Meyer Ch 3.1]

5 Rules to Ensure Modularity

1. **Direct Mapping** : The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain
2. **Few Interfaces** : Every module should communicate with as few others as possible.
3. **Small Interfaces** : If two modules communicate, they should exchange as little information as possible
4. **Explicit Interfaces** : Whenever two modules A and B communicate, this must be obvious from the text of A or B or both.
5. **Information Hiding** : The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

[See Meyer Ch 3.2]

5 Principles of Software Construction

1. **Linguistic Modular Units principle** : Modules must correspond to syntactic units in the language used.
2. **Self-Documentation principle** : The designer of a module should strive to make all information about the module part of the module itself.
3. **Uniform Access principle** : All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation.
4. **Open-Closed principle** : Modules should be both open and closed.
5. **Single Choice principle** : Whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list.

[See Meyer Ch 3.3]

Abstract Data Types (ADT)

- ADT
  - Increase cohesion
  - Separation of internal implementation from external interface
    - External use by client; client dependent on and codes only to the interface, not implementation
    - Internal definition by class designer (i.e. server); internal changes impact only class designer
    - Results in reusable, testable code
- OO = ADT +
  - Encapsulation
  - Type extension (inheritance – increases coupling)
  - Substitutability (polymorphism)

Class Construct

- As a type definition…
  - Encapsulated ADT
  - Private implementation
  - Public interface
  - External dependencies minimized
  - Supported in modern OOPL (i.e. C++, Java, C#, etc.)

- Example: `Queue`
  - Interface: `enqueue()`, `dequeue()`, `empty()`, …
  - Implementation: circular array, linked list
  - Client's use of Queue is not affected by internal implementation

Class Terminology

- A **class method** (**member function**) is a function declared in class scope
- An **instance of a class** (**object**) is an allocation/declaration of that class definition
- Member functions must distinguish between multiple instances of the class
- **this** pointer
  - Holds the address of the active object
  - An implicit parameter passed with each member function invocation
- **Internal state**
  - Value of internal data members at a given point

Object Definitions

- Self-contained entity (data & operations encapsulated)
- Instance of an ADT/class that has
  - Internal state
    - Value of all fields (properties)
    - Class should control state => minimize set/get
  - Behavior
    - Public functionality
    - State changes so trigger should be consistent
  - Identity
    - Name (static type checking)
    - Value (dynamic type checking)

OO Tenets (Principles)

- Abstraction
- Encapsulation
- Information hiding

Central class design principle: **control of state**

Abstraction

- Implementation details abstracted away
- Application programmer need not know/care how type stored or manipulated
  - Just as with built-in types
- Application programmer not responsible for
  - Initializing object
  - Maintaining consistent state of object
  - Proper manipulation
  - Bounds checking
- Incompetent/malicious programmer cannot subvert type

Encapsulation

- Wrap up type definition in one class

  (capsule) => data protected

- Low coupling
  - No extraneous functionality or data
  - No external manipulation of data
- High cohesion
  - All needed data AND functionality contained within type definition
- Maintainable

Information Hiding

- Idealization of Abstraction
  - Theory compromised by compiler's need to know size
  - Information hiding not fully realized
  - C++ (.h files contain private data declarations)
  - Java (private data and public interface in same file)
- Implementation hidden
  - Only interface is public
- Client has no knowledge of implementation details
  - Client code dependent on interface only
- Robust
  - External forces cannot put object into invalid state
- Extensible

Class Design Overview

- Type Definition
  - No memory allocated until instantiated
- Accessibility
  - public, protected, private
- Data
  - Instance data
    - exists for every object instantiated
  - Class-wide global(s)
    - Static member(s)
    - One copy for ENTIRE class
    - Independent of objects instantiated

Class Design Functionality (1)

- Constructor
  - Set object in valid, initial state
    - explicit with `new` operator; implicit in C++
  - Initialize data
  - Allocate resources
- Accessors
  - `get()` – should be `const`
  - Controlled peek inside class
- Mutators
  - `set()` – minimize
  - Controlled alteration of state
  - Check values
    - Discard out-of-bounds values
    - Provide default values

Class Design Functionality (2)

- Private utility functions
  - Support reuse within class (functional decomposition)
- Protected interface
  - Utility to be inherited by child classes
- Public interface
  - Support type definition
  - Minimum functionality needed by client
- Cleanup
  - Bookkeeping – static data members
  - C++ destructor – deallocate resources
  - Java `finalize()` – called when GC collects dead objects

Standard Class Design

- Class **design** encapsulates
  - Data and associated functionality
- Class **design** should
  - Control internal state and access to data/state transitions
- Class **methods** include
  - Constructors, accessors, mutators, private utility functions
  - For C++, possibly destructors, copy constructors, overloaded assignment
- Type driven **public functionality**
  - That which is essential to external manipulation of the type

Why Constructors?

- Constructors remove object initialization responsibility from client
- Objects ‘automatically’ placed in a valid, initial state upon instantiation
- Failure to initialize does not mean that the data has no value
  - Uninitialized data assumes whatever residual bit string value that resides in the allocated memory
  - Some languages (Java, C#) zero out fields
- A class without constructor yields objects in an indeterminate or ‘valueless’ initial state

Why Private Utility Functions?

- Provide functional decomposition
- Improve readability and maintainability (DRY)
- Encourage consistent behavior
- Encapsulation controls accessibility
  - application programmers do not have direct access to such methods

Why Minimize Set/Get?

- Control access to object's internal data
- Promote software maintainability
  - client programs to interface, not implementation
- If object's internal state known
  - Feasible to code in a manner dependent on such internal details
  - Compromise maintainability
- Accessors (gets) expose part or all of an object state
- Mutators change object state
  - should be conditional
  - class designer should determine when it is appropriate to reject a change made through a call to a mutator

When to Define Destructor?

- C++ destructors are resource managers
- Destructors release internally allocated heap memory
- Destructors may also track number of active objects, decrement reference counts, close files, etc.

Note:

- Not available in Java
  - Though a finalize() method is recommend to aid garbage collection
- Not really used in C#
  - It's complicated… (`IDisposable`)

Q & A

Next week…

- Programming fundamentals, classes, methods (and how they all fit together), exceptions, unit testing, Javadoc documentation