

CS 247 Midterm Review

ADT Design

- **Best practices**
 - All data members should be private
 - Client accesses data through public methods
 - Derived classes access data through protected methods
 - Use `const` for function parameters whenever possible
 - Accessor methods should be `const`
- **Default arguments:** only trailing parameters can have default values
 - `function (int p1, int p2 = 0, int p3 = 1);`
- **Implicit type conversion**
 - Prohibit by declaring constructor as `explicit`
- **Friends:** can access private members from outside of class
 - Often used for streaming operators
- **Operator overloading/special member functions**
 - **Default constructor:** generated by compiler if & only if no constructors are declared
 - **Destructor:** compiler default deallocates stack-based members, and calls destructors of member objects
 - **Copy constructor:** compiler default performs shallow copy
 - **Assignment operator:** compiler default performs shallow copy
 - **Equality operator:** no compiler default
 - **istream/ostream operator** (non-member functions): declare as friend
- **Rule of 3:**
 - Destructor, copy constructor, assignment operator
 - If one of these need to be defined, then all of them should be defined
- **Entity-based ADT**
 - Prohibit assignment & copy constructor
 - Prohibit type conversion
 - Compare pointer addresses for equality
 - Mutable
- **Value-based ADT**
 - Implement assignment & copy constructor
 - Implement equality & comparisons

- Immutable
- **Header guard:** prevent errors when header declarations are included multiple times
 - ```
#ifndef CLASS_H
#define CLASS_H
class Class { ...};
#endif // Class
```
- **Copy-swap idiom:** used to implement exception-safe assignment (e.g. `A = B;`)
  - Create local copy of B using copy constructor
  - Swap contents of A and new copy of B
  - A now has the contents of B; copy of B is deleted by destructor on function return
- **PImpl idiom:** instead of declaring internal workings (private data members) of a class in the public header file, declare in a nested class/struct in a separate file
  - ADT keeps a pointer to the Impl struct

## Documentation

- **Interface specification**
  - **Specification fields:** client's view of the object's fields (including private members)
  - Preconditions:
    - **Requires:** can throw exception immediately if preconditions are not satisfied
  - Postconditions:
    - **Modifies:** members that are changed
    - **Ensures:** effects on the changed members (e.g. `this = this@pre + next`)
    - **Throws:** exceptions that may be thrown, and their conditions
    - **Returns:** return value & type
  - Preconditions  $\implies$  postconditions
  - Spec A is stronger than spec B (i.e.  $A \implies B$ ) if & only if:
    - A's preconditions are equal or weaker than B
    - A's postconditions are equal or stronger than B
    - A modifies equal or more objects than B
    - A throws equal or fewer exceptions than B
- **Representation invariant**
  - A predicate in an ADT that must be true at all times
  - Structural invariants
    - e.g. two tree nodes cannot share the same child node; trees cannot have cycles
  - Value invariants

- e.g. no duplicate data elements; a value cannot be null
- Should be checked on exit of constructor and mutators
- **Abstraction function**
  - Maps concrete values to abstract values in an ADT

## Exceptions & Smart Pointers

- **Assertion:** use to check a certain condition
  - Terminates program immediately without changing its state
- **Exception:**
  - Object representing an error that can be *thrown* and *caught*
  - The call stack is popped/unwinded down to the nearest matching **catch** block
    - Destructors of all stack objects are called
    - i.e. heap objects are only handled properly if they are deleted in a stack object's destructor
- **Smart pointer:**
  - Pointer encapsulated in a stack-based object
  - Heap object is deleted in pointer object's destructor
    - i.e. if exception is raised, heap object is deleted by the pointer's destructor
  - `unique_ptr<T>`: exclusive ownership of referent; ownership can be transferred
  - `shared_ptr<T>`: shared ownership of referent
    - Object is deleted when the # of `shared_ptrs` pointing to it reaches 0
  - `weak_ptr<T>`: same as `shared_ptr`, but doesn't contribute to reference count
    - Need to check if expired, and then convert into `shared_ptr` before dereferencing

## RAII Idiom

- **Resource Acquisition is Initialization:** resource management is coupled with lifetime of object
  - Allocate resource in constructor; deallocate resource in destructor
  - `Class (param) : res_( allocate (param); ) { }`
  - `~Class() { release (res_); }`

## UML (Unified Modelling Language)

- Attributes
  - [visibility] name: [type] [multiplicity] = [default value] {property}
  - e.g. - playerName: string[1] = ""
- Operations
  - [visibility] name (parameter list) : [return type] {property}
  - e.g. + getPlayerName ( playerId : int ) : string const
  - + public; - private; # protected; static; *pure virtual*
  - property = read-only (aka. const), query (aka. accessor), abstract, etc.
- Associations: physical or conceptual links between classes
  - Classes being associated may have role names
  - Navigability: direction of association; e.g. A *has* ( $\rightarrow$ ) B
- Multiplicity: how many objects may fill the attribute/may be linked by an association
  - $a$  = exactly  $a$
  - $m..n$  = between  $m$  and  $n$  (inclusive)
  - $*$  = many (at least zero)
  - $m..*$  = at least  $m$
- Aggregate: a collection of many members
  - Member can belong to many collections, or exist independently
  - Collection is not responsible for its members
- Composition: a stricter collection of members
  - Member cannot exist without its collection
  - Member belongs to exactly one collection
  - Collection is responsible for its members
- Generalization = inheritance
- **Sequence diagrams:** describe how information is passed between objects (e.g. via function calls), throughout execution of a program
  - Good at showing how different objects collaborate; not good at defining their behaviours precisely

## OOP Principles

- **Open Closed Principle**

- Modules should be *open for extension* but *closed for modification*
- “Program to an interface, not an implementation”
- e.g. provide an abstract base class (may have default implementation) that can be extended by the client

- **Composition Over Inheritance**

- Composition = include base class in new subclass as a complex attribute
  - i.e. “has-a” instead of “is-a”
- Choose inheritance when subtyping, or when base class’s original interface is required
- Choose composition for non-overriding extension or when new required interface is different from original, because the base component can be changed at runtime
- Composite object can delegate operations to component objects (call the component’s corresponding method)

- **Single-Responsibility Principle**

- Each changeable design decision (responsibility) should be encapsulated in a module
- An axis of change is an axis of change only if the changes occur; i.e. no need to separate responsibilities if they always change at different times

- **Liskov Substitutability Principle**

- A derived class must be substitutable for its base class
  - Must accept the same messages (method signatures match the base class)
  - Derived methods must require no more (weaker or same preconditions) and promise no less (stronger or same postconditions) than base class methods
    - ◇ Method return types match (or be a subtype) the base class
  - Derived class must preserve properties of base class (e.g. invariant, performance)

- **Law of Demeter**

- An object should only “talk to its neighbours” (component composition)
- A method `C::m()` can only call methods of:
  - C
  - C’s members
  - m’s parameters
  - Any object constructed by A’s methods
- Bad: `game->getPlayer()->getStatus()->getHP();`
- Good: `game->getPlayerHP();`

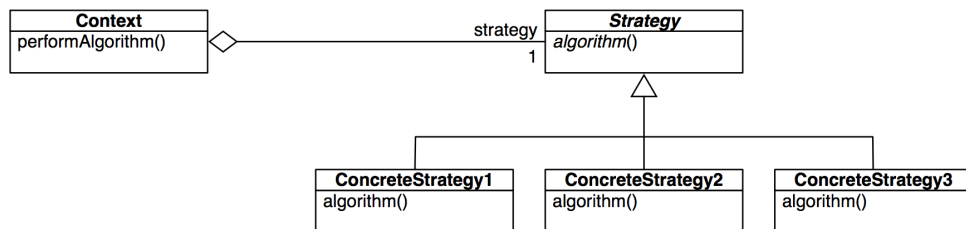
## Design Patterns

- **Inheritance**

- Parent class's methods are inherited by child classes
  - Classes' methods share the same implementation structure
  - Only differences are the data values
- Downside: not all subclasses may want to inherit parent behaviour
- Downside: code duplication

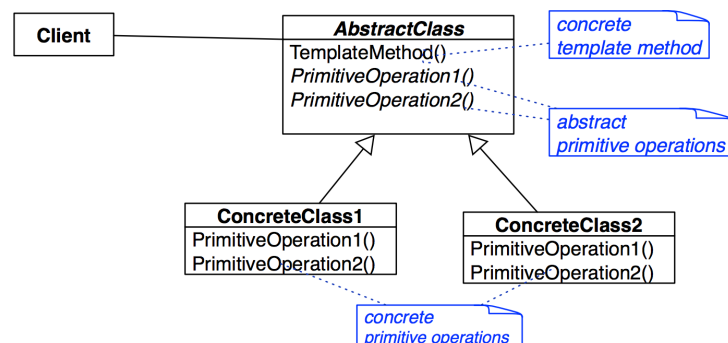
- **Strategy pattern**

- Allows the implementation of an algorithm/method to be changed at runtime (encapsulation of algorithm)
- Allows the algorithm to vary independently from clients that use it
- e.g. data structure holds an instance of base Strategy class, calls the algorithm/method (which is *pure virtual* in base class)
  - Concrete methods with differing behaviour are implemented in Strategy subclasses
  - Strategy can be changed (to other subclasses) at runtime, changing the method's behaviour



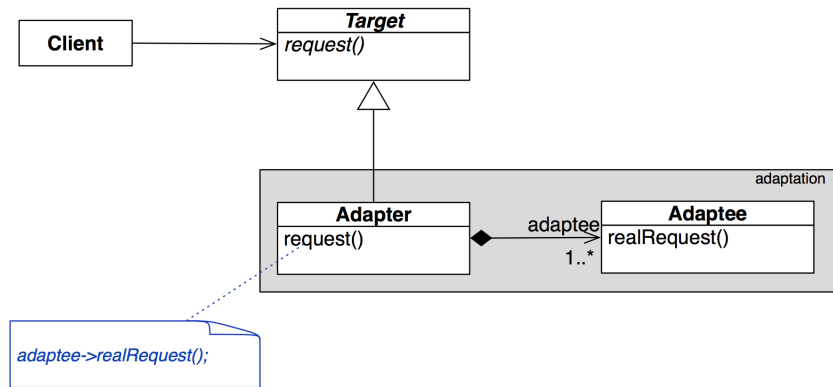
- **Template pattern**

- **Template method** is a method in a base class that defines code structure but leaves holes to be defined by subclasses
- Holes are operations defined as *pure virtual* in the base class, but have varying implementations in subclasses



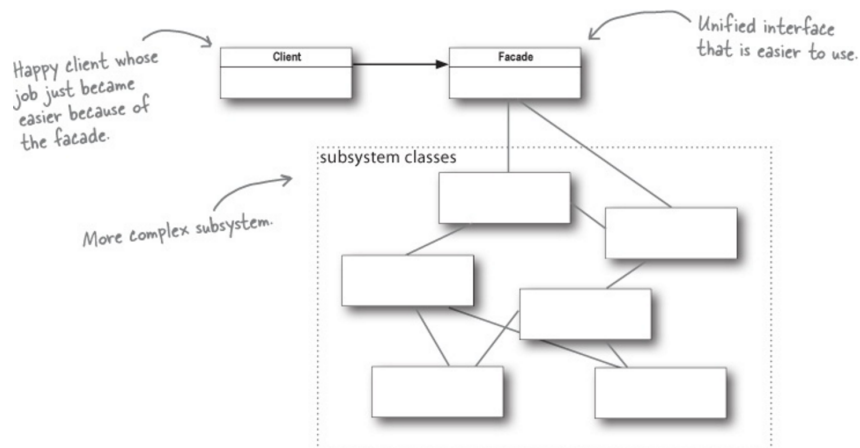
- **Adapter pattern**

- Adapter maps one interface to another
- e.g. interfaces of an existing module does not match with a new module
- e.g. wrapping an existing data structure interface to create a new data structure



- **Facade pattern**

- Simplifies and unifies classes and interfaces in a subsystem into only a high-level interface and hides individual interfaces within the subsystem
- Subsystem components and interfaces can be changed without affecting client

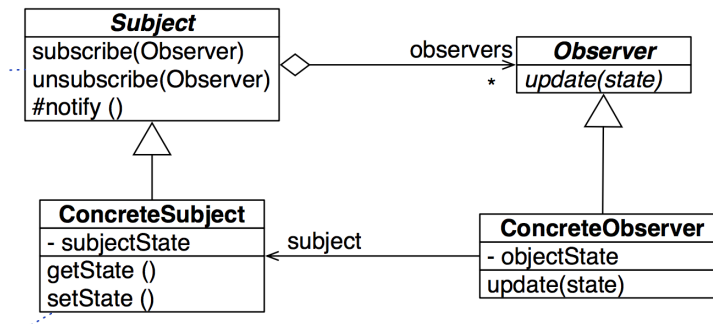


- **Singleton pattern**

- Ensures only one instance of a class can exist
- Private constructor; only instantiated through static `getInstance()` method

- **Observer pattern**

- Subject  $\rightarrow$  (one-to-many) Observers
- Subject can notify all subscribed observers to update
- Observers can subscribe/unsubscribe at runtime
- **Push model**: subject pushes state information to observers through `notify(State)`
- **Pull model**: subject notifies observers, who request information via subject's accessors
- **Loose coupling**: subjects and observers only know about each other's interfaces, not the concrete classes that implement them



- **MVC pattern**

- UI code is abstracted into the **view**
  - Composite pattern: all view elements use the same base class (uniform interface)
- **Controller** translates user input (from the view) into operations on the model
  - Strategy pattern: controller provides the view with a strategy; controller behaviour can be changed by swapping for a different strategy
- **Model** holds data, state, and application logic
  - Observer pattern: model = subject; views = observers; model sends out notification on state change, triggering views to update accordingly