

CS 247 Midterm Review

ADT Design

- Bullet point here

Documentation

Exceptions

RAII Idiom

UML

- **Unified Modelling Language**
- Class diagrams
 - Attributes
 - `[visibility] name: [type] [multiplicity] = [default value] {property}`
 - Operations
 - `[visibility] name (parameter list) : [return type] {property}`
 - `+ 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 B*
- Multiplicity (of attributes or associations)
 - *a*: exactly *a*
 - *m..n*: between *m* and *n*
 - ***: many (at least zero)
- Aggregate: a collection of members
 - Collection has 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

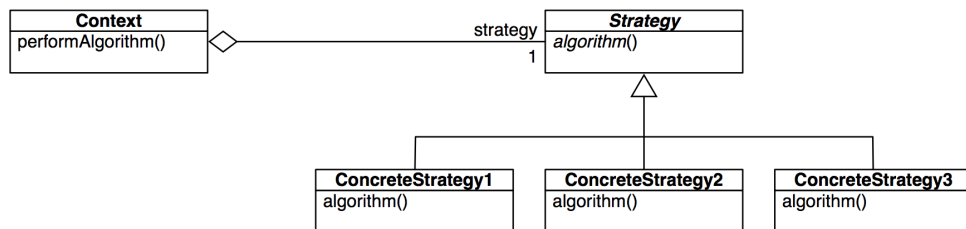
Design Patterns

- **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

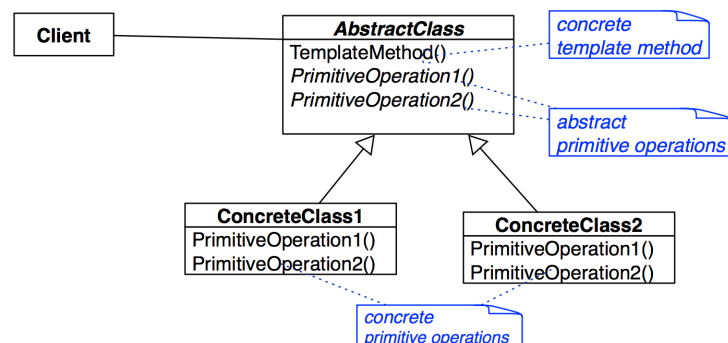
- **Strategy pattern**

- Allows the implementation of an algorithm/method to be changed at runtime (encapsulation of algorithm)
- Allows the algorithm 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



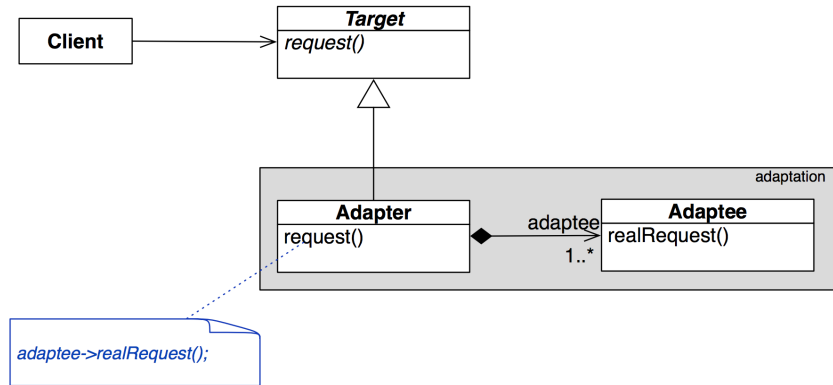
- **Template pattern**

- **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



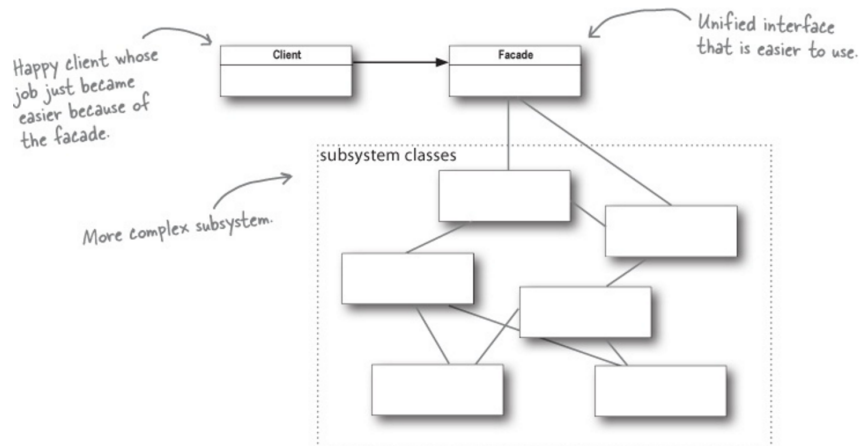
- **Adapter pattern**

- 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

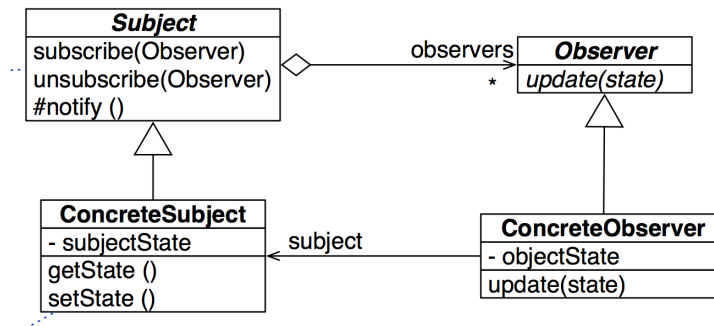


- **Singleton pattern**

- 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

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/polymorphism, or when base class’s entire interface is required
- Choose composition for non-overriding reuse because the base component can be changed at runtime
- Composite object can delegate operations to component objects

- **Single-Responsibility Principle**

- Each changeable design decision should be encapsulated in a module

- **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
 - Derived class must preserve properties of base class (e.g. invariant, performance)

- **Law of Demeter**

- An object should only “talk to its neighbours”
- A method `C::m()` can only call methods of:
 - C
 - C’s members
 - m’s parameters
 - Any object constructed by A’s methods
- Prevents calling a chain of methods to perform an operation/retrieve information