Design issues:
abstraction and patterns
Strategy Pattern
Singleton Pattern

CSIE Department, NTUT
Woei-Kae Chen
Common Mistake (C)
- no abstraction at all \implies ADT

```c
typedef struct _X {
    int x;
} X;

void inc(X *a) {
    a.x++;
}
```

```c
... X a;
    a.x++;

ADT
```
Common Mistake (C++)
- not enough abstraction

```cpp
class X {
public:
    void SetX(int _x) {x = _x;}
    int  GetX() {return x;}
private:
    int x;
}
```

```cpp
X a;
a.SetX(a.GetX()+1);
```
class X
{
public:
    void inc() {x++;}
    int GetX() {return x;}
private:
    int x;
}

X a;
a.inc();
class Token // Abstraction
{
    public:
    int init(int _n, int _limit)
    {n = _n; limit = _limit;}
    void take(int d) {n -= d;}
    void put(int d) {n += d;}
    bool isEnough(int d)
    {return ((n-d)>=0);}
    bool isFull(int d)
    {return ((n+d)<=limit);}

    private:
    int n, limit;
};
class Transition // Abstraction
{
public:
    bool isEnabled() ...
    void fire() ...
    void addInputPin(Place *) ...
    void addOutputPin(Place *) ...
    ...
    bool deleteInputPin(Place *) ...
    bool deleteOutputPin(Place *) ...
};
class Elevator // Abstraction
{
public:
    ... PressOpenDoorButton() ...
    ... PressCloseDoorButton() ...
    ... PressTargetFloorButton() ...
    ... Move() ...
    ... GetFloor() ...
private:
    ... OpenDoor() ...
    ... CloseDoor() ...
    ... doorStatus ...
    ... floorNum ...
}
Experience: OOSortX

Strategy pattern: Define a family of algorithms, encapsulate each one, and make them interchangeable.
Design patterns

- simple and elegant solutions to specific problems in object-oriented software design
- capture solutions that have been developed and evolved over time
- make your own designs more
  - flexible,
  - modular,
  - reusable, and
  - understandable
Strategy pattern: Intent

- Define a family of algorithms.
  - Encapsulate each one (as an object).
  - Make them interchangeable (through polymorphism).
  - Vary independently from clients that use it

- Note: OMT ➔ page 365
## Strategy pattern: Example

<table>
<thead>
<tr>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleComposition()</td>
</tr>
<tr>
<td>TexComposition()</td>
</tr>
<tr>
<td>ArrayComposition()</td>
</tr>
</tbody>
</table>
Strategy pattern: Motivation

- Hard-wiring line breaking algorithms
  - Clients get bigger and harder to maintain.
  - Different algorithms will be appropriate at different times (must support all).
  - Difficult to add new algorithms and vary existing ones.

- Strategy avoids these problems
Strategy pattern: Example

Composition
- Traverse ()
- Repair ()

Compositor -> Compose ()

Compose ()

SimpleCompositor
- Compose ()

TexCompositor
- Compose ()

ArrayCompositor
- Compose ()

line breaking
Strategy pattern: Applicability

- Many related classes differ only in their behavior.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about. Avoid exposing complex, algorithm-specific data structure to clients.
- A class defines many behaviors (use of multiple conditional statements).
Strategy pattern: Structure

- **Context**
  - +ContextInterface()

- **Strategy**
  - +AlgorithmInterface()

- **Client**

- **ConcreteStrategyA**
  - +AlgorithmInterface()

- **ConcreteStrategyB**
  - +AlgorithmInterface()

- **ConcreteStrategyC**
  - +AlgorithmInterface()
Strategy pattern: Participants

- **Strategy (Compositor)**
  - Declares an interface common to all supported algorithms.

- **ConcreteStrategy (SimpleCompositor ...)**
  - Implements the algorithm using the strategy interface.

- **Context (Composition)**
  - Is configured with a ConcreteStrategy object.
  - Maintains a reference to a Strategy object.
  - May define an interface that lets Strategy access its date.
Strategy pattern: Collaborations

- **Passing data**
  - A context may pass all data required by the algorithm.
  - Alternatively, the context can pass itself as an argument to Strategy operations.

- **A context forwards requests from its clients to its strategy.**
Strategy pattern: Benefits

- Families of related algorithms.
- An alternative to subclassing (subclassing context).
- Strategies eliminate conditional statements.
- A choice of implementations (client can choose different strategies).
Strategy pattern: Drawbacks

- Clients must be aware of different Strategies.
- Communication overhead between Strategy and Context.
  - Won’t use all the information passed to them.
  - Solution: Tighter coupling between Strategy and Context.
- Increased number of objects.
Strategy pattern: Implementation

- Defining the Strategy and Context interfaces.
  - Context pass data to Strategy operations.
    - Decoupled.
    - Might pass unneeded data.
  - Context passed itself as an argument.
    - Strategy requests exactly what it needs.
    - Must define a more elaborate interface.
    - Couples.
Strategies as template parameters.
- The strategy can be selected at compile-time.
- It does not have to be changed at run-time.

```
template <class AStrategy>
class Context {
  void Operation()
  { theStrategy.DoAlgorithm(); }
  // ...
private:
  Astrategy theStrategy;
};
```

- No abstract Strategy class.
- Making Strategy objects optional (context carries default behavior).

```
class MyStrategy {
  public:
    void DoAlgorithm();
};
Context<MyStrategy> myStrategyContext;
```
Singleton pattern: intent

- Ensure a class only has one instance, and provide a global point of access to it.

Example:
- clock,
- printer spooler
- file system
- window manager
- A/D converter
- I/O port
Singleton pattern: motivation

- How do we ensure that a class has only one instance and the instance is easily accessible?
  - global variable
    - makes an object accessible
    - but cannot ensure from multiple objects
  - make the class itself responsible
    - ensure no other instance can be created
    - provide a way to access the instance
Singleton pattern: structure

**Singleton**
- `static Instance()`
- `SingletonOperation()`
- `GetSingletonData()`
- `static uniqueInstance`
- `singletonData`

**Diagram:**
- Return unique instance
- Static member variables
- Static member functions
Singleton pattern: applicability

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
Singleton pattern: participants

- Singleton
  - Defines an Instance operation that lets clients access its unique instance.
  - May be responsible for creating its own unique instance.
Singleton pattern: Collaborations

- Clients access a Singleton instance solely through Singleton’s Instance operation.
Singleton pattern: Consequences

- Controlled access to sole instance.
- Reduced name space (no pollution to the name space of global variables).
- Permits refinement of operations and representation (may be subclassed).
- Permits a variable number of instances.
- More flexible than class operations (C++ static member functions).
  - more than one instance allowed
  - subclassing allowed
Singleton pattern: Implementation (1)

Ensuring a unique instance.

- protected constructor
- * _instance

```cpp
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
}

Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance() {
    if(_instance==0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Who: delete _instance? (memory leak)
Singleton pattern: Implementation (2)

- C++ global or static object
  - can’t guarantee single instance
  - automatic initialization may not work (a single might require values that are computed after initialization)
  - order of constructors of global objects are not defined (no dependencies can exist between singletons)
  - created whether they are used or not
Singleton pattern: Implementation (3)

Subclassing the Singleton class.

class Singleton {
  public:
    static void Register(char* name, Singleton*);
    static Singleton* Instance();
  protected:
    static Singleton* Lookup(const char* name);
  private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry; }

Singleton pattern: Implementation (4)

```
Singleton* Singleton::Instance() {
    if(_instance==0) {
        const char* singletonName = getenv("SINGLETON");
        _instance = Lookup(singletonName);
    }
    return _instance;
}

MySingleton::MySingleton() {
    Singleton::Register("MySingleton", this);
}

static MySingleton theSingleton;
```