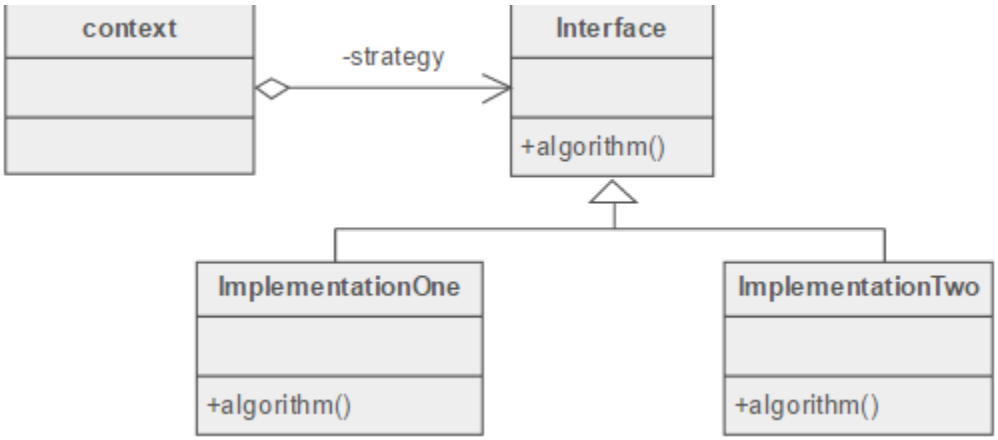
STRATEGY PATTERN

1/ Introduction

- If we handle task which often changes, having to edit all those classes becomes a maintenance issue.
- If we use inheritance, we will spread out the handling of a changeable task over several generations of classes. We are going to maintain a lot of customized code across generations of classes when that task changes. And as the derived classes get to be long and involved, it's going to be tough to maintain them through all those changes.
- Getting to Know Strategy Pattern is a better way of handling classes differ only in their behaviors
- In strategy pattern, define a family of algorithm, encapsulates each one and make them interchangeable. Strategy lets the algorithm vary independently from context that use it.



Strategy - defines an interface for all supported algorithms. Context uses this interface to call the algorithm.

2/Problem

Our company receives transportation manufacturing from the government. For each vehicle there are different control method (car, motorbike-drive, plane-fly) and different indexes on velocity and wheel size.

3/Usual Way

Using inheritance to implement vehicle as base class and different types of methods as derived class. In these classes, use a function to stimulate the attribute of each vehicle.

```
class Vehicle
{
public:
    Vehicle() {};
    virtual void go() = 0;
};
```

4/Issues with usual way

At first, there is nothing serious the inheritance solution. Later if there are enormous changes on details of the vehicle. however, the maintenance on the codes which need changing is somehow complicate and not sufficient. For the example above there is nothing hard as the code is simple but for the big project this is a complex process.

```
class Car : public Vehicle
{
public:
    Car() {};
    void go()
    {
        cout << "Driving" << endl;
    }
};

class Plane : public Vehicle
{
public:
    Plane() {};
    void go()
    {
        cout << "Flying" << endl;
    }
};
```

5/Strategy pattern

- In design pattern terms, each implementation of the *go* method is called an *algorithm*.
- Separate the code the code to be changed into algorithms.
- Each algorithm handles one complete task. Therefore, we don't have to spread out the handling of that task over generations of classes.

```
class GoAlgorithm
{
public:
    virtual void go() = 0;
};
```

```
class GoByDrivingAlgorithm : public GoAlgorithm
{
public:
    virtual void go()
    {
        cout << "Driving" << endl;
    }
};

class GoByFlyingAlgorithm : public GoAlgorithm
{
public:
    virtual void go()
    {
        cout << "Flying" << endl;
    }
};
```

```
class Vehicle
{
private:
    GoAlgorithm *v_go;
public:
    void setGo(GoAlgorithm *x)
    {
        v_go = x;
    }
    void go()
    {
        v_go->go();
    }
};
```

6/Similar problem

- ❖ Stimulating how searching and sorting work in binary tree
- ❖ SearchBehavior and SortBehavior is separated as algorithms from Tree class.

Code:

```
class SortBehavior
{
public:
    virtual void sort() const = 0;
};
```

```
class Merge : public SortBehavior
{
public:
    virtual void sort() const
    {
        cout << "Merge sort()\n";
    }
};

class Quick : public SortBehavior
{
public:
    virtual void sort() const
    {
        cout << "Quick sort()\n";
    }
};
```

(Sort algorithm)

```
class SearchBehavior
{
public:
    virtual void search() const = 0;
};
```

```
class Sequential : public SearchBehavior
{
public:
    virtual void search() const {
        cout << "Sequential search()\n";
    }
};

class BinaryTree : public SearchBehavior
{
public:
    virtual void search() const {
        cout << "BinaryTree search()\n";
    }
};
```

(Search algorithm)

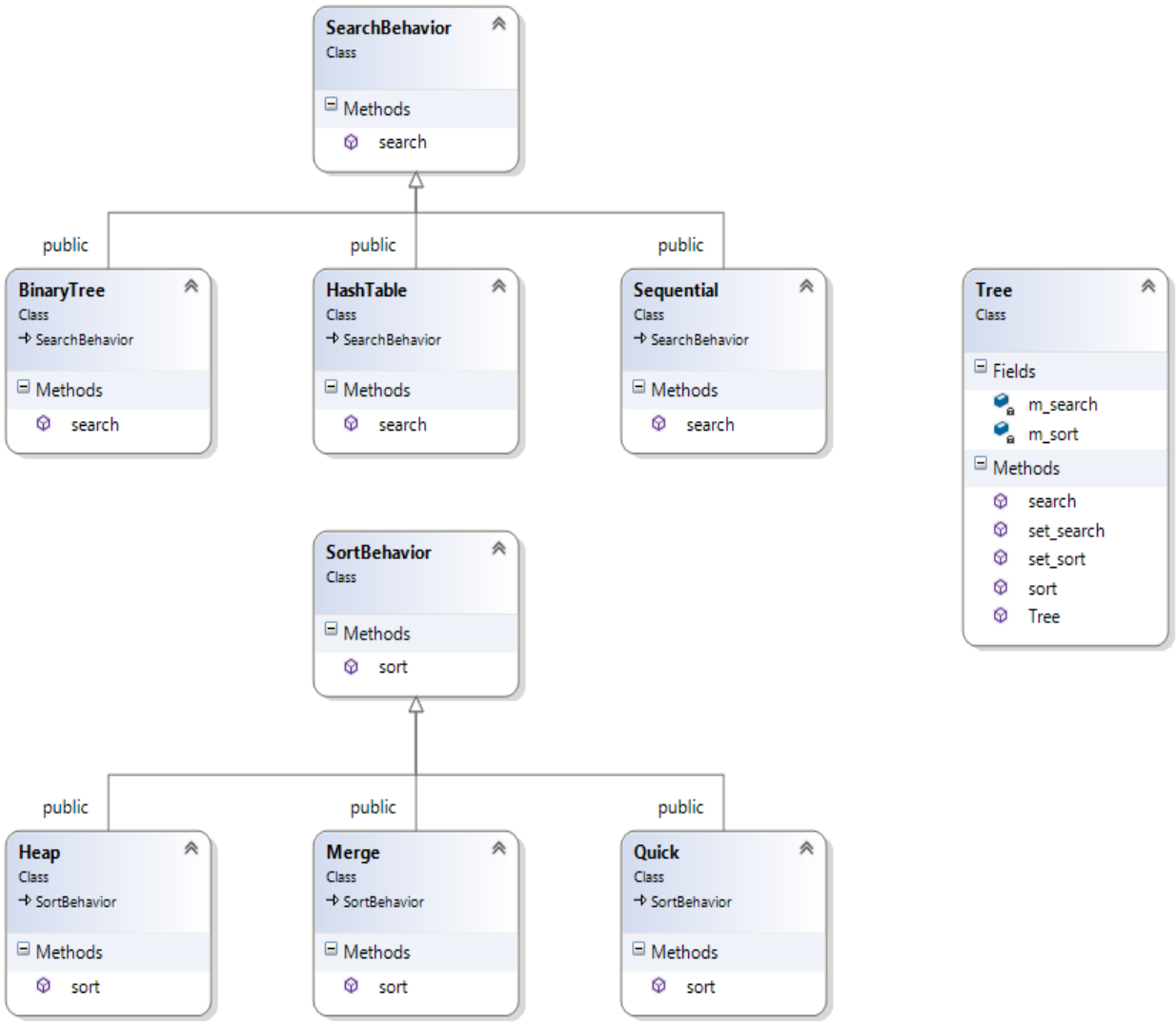
```
class Context
{
private:
    SortBehavior* m_sort;
    SearchBehavior* m_search;
public:
    Context() {}
    void set_sort(SortBehavior* s) {
        m_sort = s;
    }
    void set_search(SearchBehavior* s) {
        m_search = s;
    }
    void sort() const {
        m_sort->sort();
    }
    void search() const {
        m_search->search();
    }
};
```

(The Context class contains 2 algorithms: sort and search)

```
Merge merge;

BinaryTree binaryTree;

Context conA;
conA.set_sort(&merge);
conA.sort();

Context conB;
conB.set_search(&binaryTree);
conB.search();
```

(Using context to decide which algorithm to use)

Class Diagram:



7/Note when to use

- If we have code needs to be changed we should separate out of the application for easy maintenance.
- When we want to avoid splitting implementation code over several inherited classes.
- When we want to change the algorithm at runtime.

8/Pros and Cons

8.1/Pros

- Reusability of algorithm.
- Encapsulating the algorithm in separate Strategy classes lets users vary the algorithm independently of its context, making it easier to switch, understand, and extend.
- Strategies eliminate conditional statements. The Strategy pattern offers an alternative to conditional statements for selecting desired behavior. When different behaviors are placed in one class, it's hard to avoid using conditional statements to select the right behavior. Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.
- A choice of implementations. Strategies can provide different implementations for same behavior.
- Change the behavior on the object at runtime by using algorithms.

8.2/Cons

- Increase small objects in the application. Strategies increase the number of objects in an application.
- Context must be aware of different Strategies. The pattern has a potential drawback in that a context must understand how strategies differ before it can select the appropriate one Therefore we should use the Strategy pattern only when the variation in behavior is relevant to context.
- The Strategy interface is shared by all concrete strategies (Algorithm) classes whether the algorithms they implement are trivial or complex. It's likely that some strategies won't use

all the information passed to them through this interface. Simple ones may use none of it. That means there will be times when the context creates and initializes parameters that never get used.