

# Behavioural Design Patterns

## Part 2

# Behavioural Design Patterns

- State
- Strategy
- Observer
- Command
- Visitor



# Behavioural Patterns: Strategy

- Suppose we are creating a `Date` class that can store a date/time value
- We want to provide a `toString` method that can output the `Date` in various formats ...

# Behavioural Patterns: Strategy

Date.h

```
class Date
{
public:
    enum DateFormat { DATE, TIME, DATETIME };

    Date(int, int, int, int, int, int);
    const std::string toString(DateFormat) const;

private:
    int _year;
    int _month;
    int _day;
    int _hour;
    int _minute;
    int _second;
};
```

# Behavioural Patterns: Strategy

## Date.cpp

```
const string Date::toString(DateFormat format) const {
    ostringstream os;

    switch (format) {
        case DATE:
            os << setw(2) << setfill('0') << _month << "-"
               << setw(2) << setfill('0') << _day << "-"
               << _year;

            return os.str();
            break;
        case TIME:
            os << setw(2) << setfill('0') << _hour << ":"
               << setw(2) << setfill('0') << _minute << ":"
               << setw(2) << setfill('0') << _second;
            return os.str();
            break;
        case DATETIME:
            os << setw(2) << setfill('0') << _month << "-"
               << setw(2) << setfill('0') << _day << "-"
               << _year << " "
               << setw(2) << setfill('0') << _hour << ":"
               << setw(2) << setfill('0') << _minute << ":"
               << setw(2) << setfill('0') << _second;
            return os.str();
            break;
    }
}
```

# Behavioural Patterns: Strategy

## **Design Pattern:**

### **Strategy**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

# Behavioural Patterns: Strategy

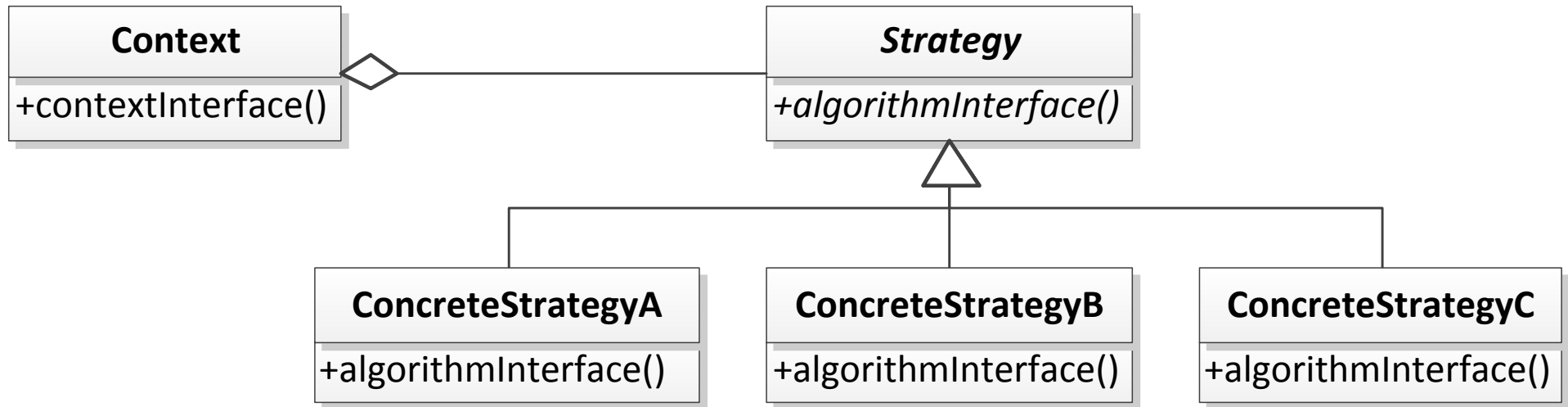
- Applicability:
  - Many related classes differ only in their behaviour; strategies provide a way to configure a class with one of many behaviours
  - You need different variants of an algorithm; for example, we might define algorithms reflecting different space/time tradeoffs

# Behavioural Patterns: Strategy

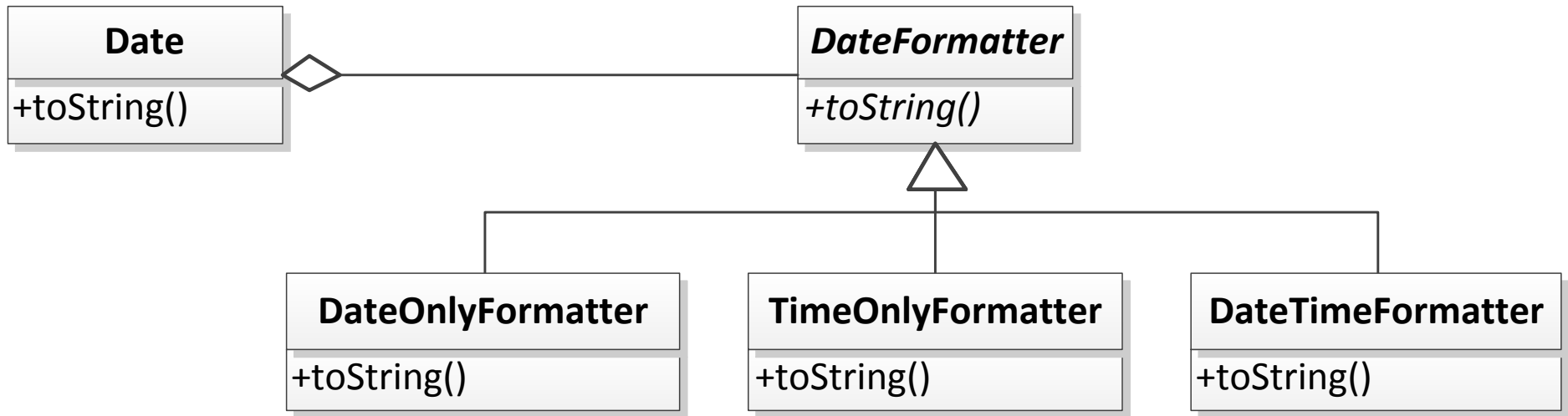
- Applicability:
  - An algorithm uses data that clients shouldn't know about; use the Strategy pattern to avoid exposing complex, algorithm-specific data structures
  - A class defines many behaviours, and these appear as multiple conditional statements in its operations; instead of many conditionals, move related conditional branches into their own Strategy classes



# Behavioural Patterns: Strategy



# Behavioural Patterns: Strategy



# Behavioural Patterns: Strategy

## Date.h

```
class Date
{
public:
    Date(int, int, int, int, int, int, DateFormatter*);

    void setFormatter(DateFormatter*);
    const std::string toString() const;

    int year() const;
    int month() const;
    int day() const;
    int hour() const;
    int minute() const;
    int second() const;

private:
    int _year;
    int _month;
    int _day;
    int _hour;
    int _minute;
    int _second;
    DateFormatter* _formatter;
};
```

# Behavioural Patterns: Strategy

Date.cpp

```
void Date::setFormatter(DateFormatter* formatter)
{
    delete this->_formatter;
    this->_formatter = formatter;
}

const string Date::toString() const
{
    return this->_formatter->toString(this);
}
```

# Behavioural Patterns: Strategy

DateFormatter.h

```
class DateFormatter
{
    public:
        virtual const std::string toString(const Date* date) const = 0;
};
```

# Behavioural Patterns: Strategy

DateOnlyFormatter.cpp

```
const std::string DateOnlyFormatter::toString(const Date* date) const
{
    ostreamstream os;

    os << setw(2) << setfill('0') << date->month() << "-"
        << setw(2) << setfill('0') << date->day() << "-"
        << date->year();

    return os.str();
}
```

# Behavioural Patterns: Strategy

DateTimeFormatter.cpp

```
const std::string DateTimeFormatter::toString(const Date* date) const
{
    ostringstream os;

    os << setw(2) << setfill('0') << date->month() << "-"
       << setw(2) << setfill('0') << date->day() << "-"
                                   << date->year() << " "
       << setw(2) << setfill('0') << date->hour() << ":"
       << setw(2) << setfill('0') << date->minute() << ":"
       << setw(2) << setfill('0') << date->second();

    return os.str();
}
```

# Behavioural Patterns: Strategy

main.cpp

```
main()
{
    Date d(2011, 11, 5, 9, 52, 0, new DateOnlyFormatter);
    cout << "Date      : " << d.toString() << endl;

    d.setFormatter(new TimeOnlyFormatter);
    cout << "Time       : " << d.toString() << endl;

    d.setFormatter(new DateTimeFormatter);
    cout << "DateTime  : " << d.toString() << endl;
}
```



# Behavioural Patterns: Strategy

## Output

Date : 11-05-2011

Time : 09:52:00

DateTime : 11-05-2011 09:52:00

# Behavioural Patterns: Strategy

- Consequences:
  - Families of related algorithms
  - Inheritance can help factor out common functionality of the algorithms
  - An alternative to subclassing
  - Eliminate conditional statements
  - A choice of implementations
  - Clients must be aware of different strategies
  - Increased number of objects