Behavioural Design Patterns

Behavioural Design Patterns

- Concerned with:
  - Algorithms
  - The assignment of responsibilities between objects

- Two types:
  - Class Behavioural - Use inheritance to distribute behaviour between classes
  - Object Behavioural - Use object composition rather than inheritance

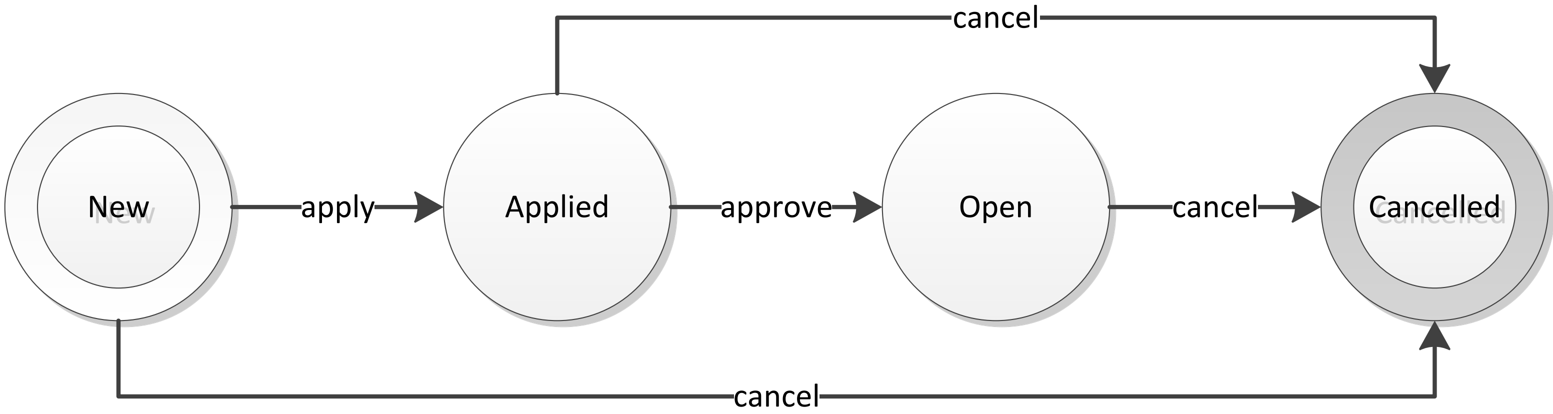
Behavioural Design Patterns

- State
- Strategy
- Observer
- Command
- Visitor

Behavioural Patterns: State

- Suppose we are building a `LineOfCredit` class
- A line of credit can be in various states:
  - New
  - Applied
  - Open
  - Cancelled
- A line of credit has various behaviours:
  - `apply`
  - `withdraw`
  - `makePayment`
  - `cancel`
- Behaviours may change depending on the current state

Behavioural Patterns: State



Behavioural Patterns: State

LineOfCredit.h

```
class LineOfCredit
{
  public:
    enum AccountState { NEW, APPLIED, OPEN, CANCELLED };

    LineOfCredit();

    const std::string state() const;
    float balanceOwing() const;
    float availableCredit() const;

    void apply(float amount);
    void approve();
    void withdraw(float amount);
    void makePayment(float amount);
    void cancel();

  private:
    AccountState _state;
    float _availableCredit;
    float _balanceOwing;
};
```

Behavioural Patterns: State

LineOfCredit.cpp

```
LineOfCredit::LineOfCredit()
{
  this->_state = NEW;
}

const string LineOfCredit::state() const
{
  switch (this->_state)
  {
    case NEW:
      return "New";
    case APPLIED:
      return "Applied";
    case OPEN:
      return "Open";
    case CANCELLED:
      return "Cancelled";
    default:
      return "Unknown";
  }
}
```

Behavioural Patterns: State

LineOfCredit.cpp

```
void LineOfCredit::apply(float amount)
{
  if (this->_state == NEW)
  {
    this->_state = APPLIED;
    this->_availableCredit = amount;
  }
  else
    throw "Can't apply in the current state";
}
```

Behavioural Patterns: State

LineOfCredit.cpp

```
void LineOfCredit::cancel()
{
  switch (this->_state)
  {
    case NEW:
    case APPLIED:
      this->_state = CANCELLED;
      break;

    case OPEN:
      if (this->_balanceOwing > 0)
        throw "If only life worked that way.";
      else
        this->_state = CANCELLED;
      break;

    default:
      throw "Can't cancel the line of credit in the current state";
      break;
  }
}
```

Behavioural Patterns: State

Design Pattern:

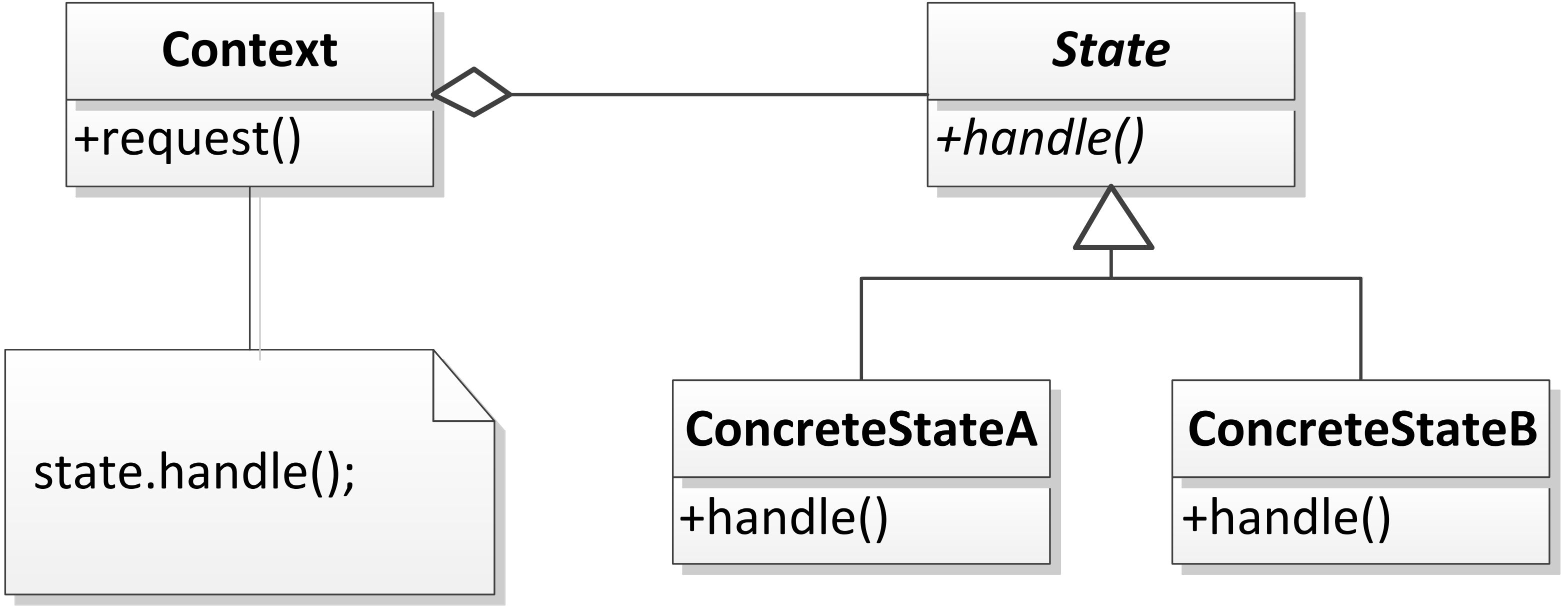
State

Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.

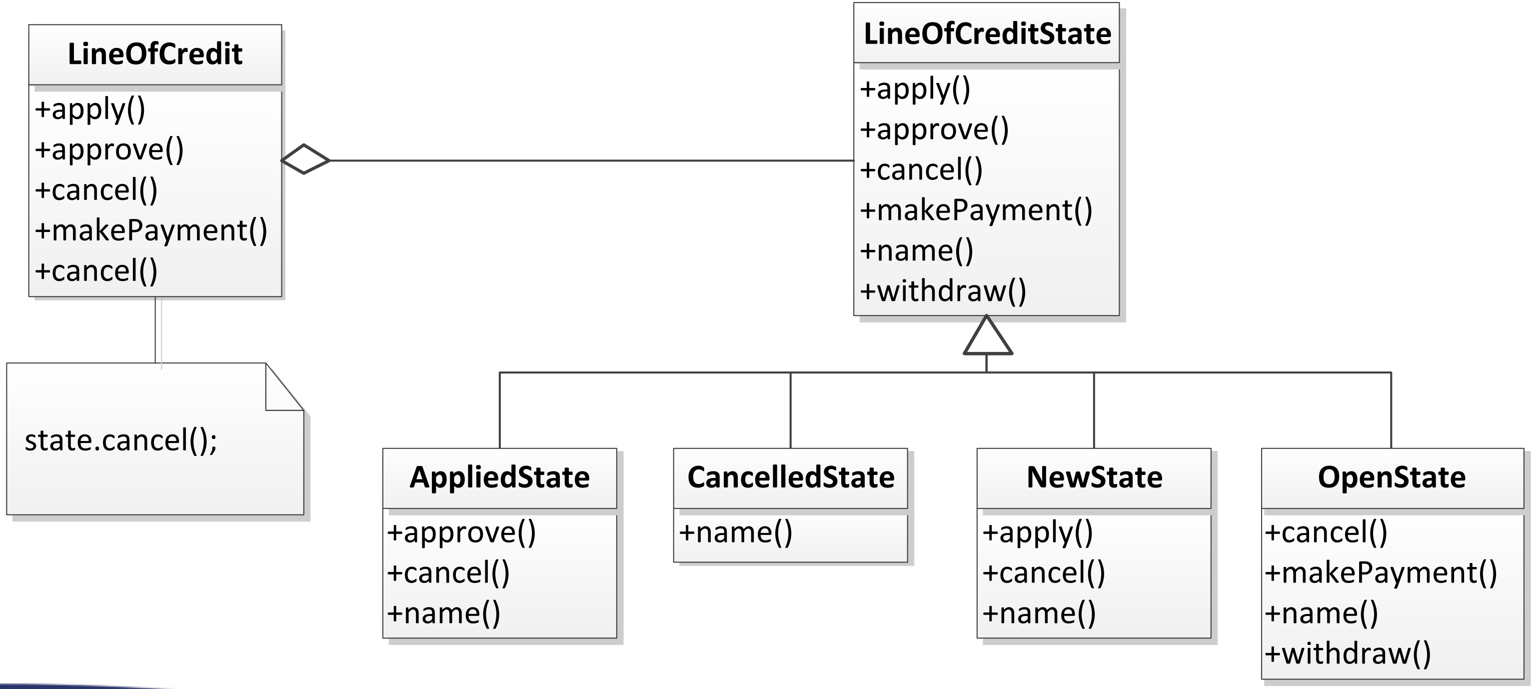
Behavioural Patterns: State

- Applicability:
  - An object’s behaviour depends on its state, and it must change its behaviour at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object’s state
    - Usually represented by one or more enumerated constants
    - Often, several operations will contain this same conditional structure

Behavioural Patterns: State



Behavioural Patterns: State



Behavioural Patterns: State

LineOfCredit.h

```
public:

    friend class LineOfCreditState;

    LineOfCredit();
    ...

private:
    LineOfCreditState* _state;
    float _availableCredit;
    float _balanceOwing;
```

Behavioural Patterns: State

LineOfCredit.cpp

```
LineOfCredit::LineOfCredit()
{
  this->_state = new NewState(this);
}

const string LineOfCredit::state() const
{
  return this->_state->name();
}
```

Behavioural Patterns: State

LineOfCredit.cpp

```
void LineOfCredit::apply(float amount)
{
  this->_state->apply(amount);
}

void LineOfCredit::approve()
{
  this->_state->approve();
}

void LineOfCredit::withdraw(float amount)
{
  this->_state->withdraw(amount);
}

void LineOfCredit::makePayment(float amount)
{
  this->_state->makePayment(amount);
}
```

Behavioural Patterns: State

LineOfCreditState.h

```
class LineOfCreditState
{
  public:
    LineOfCreditState(LineOfCredit*);
    virtual void apply(float);
    virtual void approve();
    virtual void withdraw(float);
    virtual void makePayment(float);
    virtual void cancel();
    virtual const std::string name() const;

  protected:
    LineOfCredit* _loc;
};
```

Behavioural Patterns: State

LineOfCreditState.cpp

```
LineOfCreditState::LineOfCreditState(LineOfCredit* loc) : _loc(loc)
{
}

void LineOfCreditState::apply(float amount)
{
  throw "Cannot apply in the current state";
}

void LineOfCreditState::approve()
{
  throw "Cannot approve line of credit in the current state";
}

void LineOfCreditState::withdraw(float amount)
{
  throw "Cannot withdraw from line of credit in the current state";
}
```

Behavioural Patterns: State

AppliedState.cpp

```
AppliedState::AppliedState(LineOfCredit* loc) : LineOfCreditState(loc)
{
}

void AppliedState::approve()
{
  this->_loc->_state = new OpenState(this->_loc);
}

void AppliedState::cancel()
{
  this->_loc->_state = new CancelledState;
}

const string AppliedState::name() const
{
  return "Applied";
}
```

Behavioural Patterns: State

OpenState.cpp

```
OpenState::OpenState(LineOfCredit* loc) : LineOfCreditState(loc)
{
}

void OpenState::withdraw(float amount)
{
  if (this->_loc->_balanceOwing + amount > this->_loc->_availableCredit)
    throw "Insufficient funds available";
  else
    this->_loc->_balanceOwing += amount;
}

void OpenState::makePayment(float amount)
{
  this->_loc->_balanceOwing -= amount;
}
```

Behavioural Patterns: State

OpenState.cpp

```
void OpenState::cancel()
{
  if (this->_loc->_balanceOwing > 0)
    throw "If only life worked that way.";
  else
    this->_loc->_state = new CancelledState;
}

const string OpenState::name() const
{
  return "Open";
}
```

Behavioural Patterns: State

- Consequences:
  - Localizes state-specific behaviour and partitions behaviour for different states
  - Makes state transitions explicit
  - State objects can be shared

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.

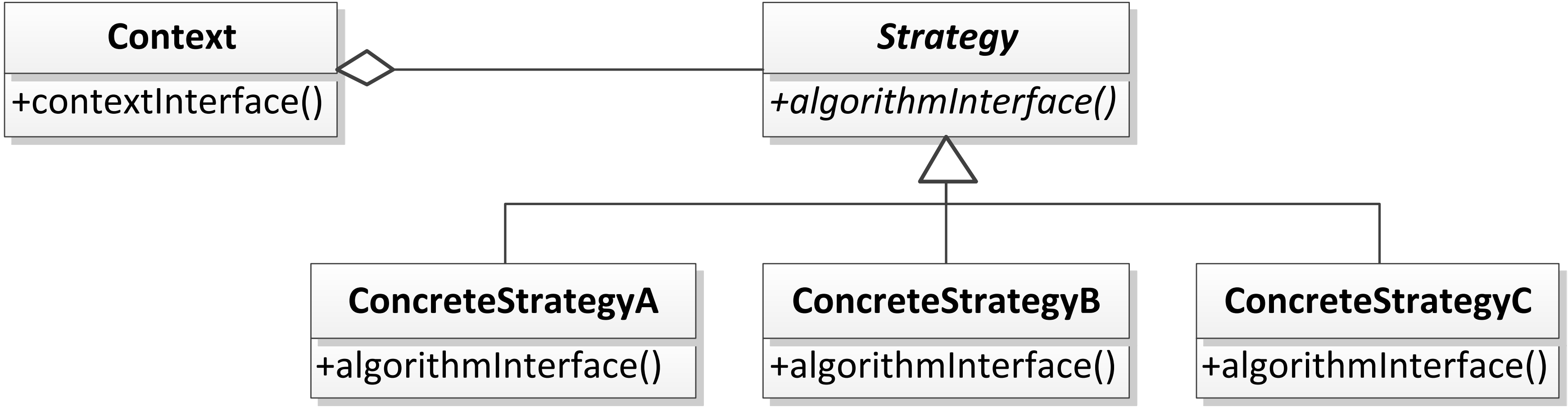
Behavioural Patterns: Strategy

- 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

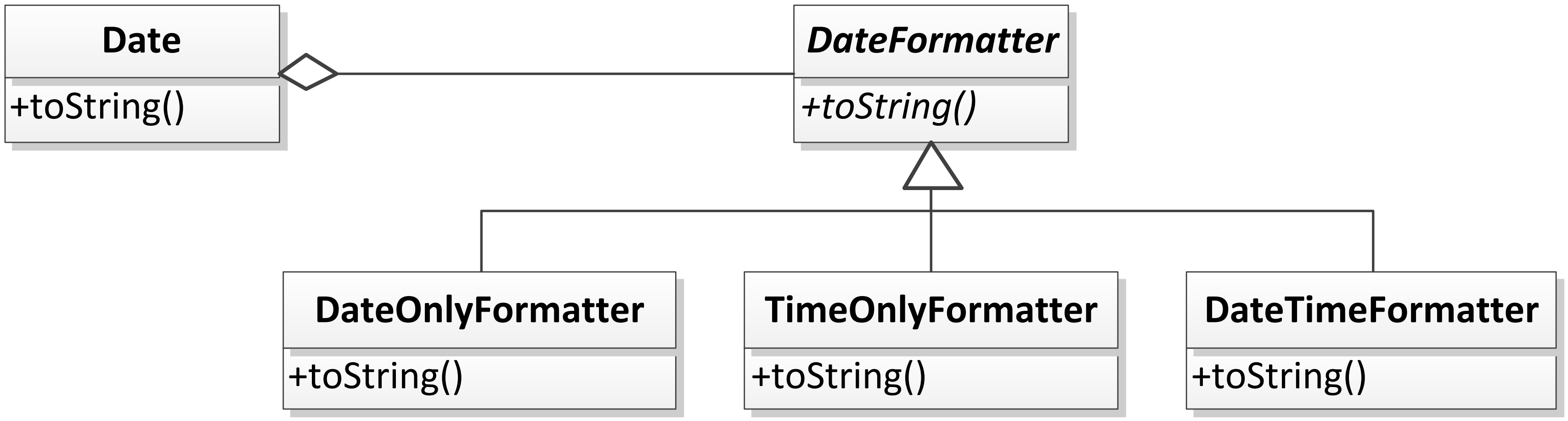
Behavioural Patterns: Strategy

- 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
{
  ostringstream 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

Behavioural Design Patterns

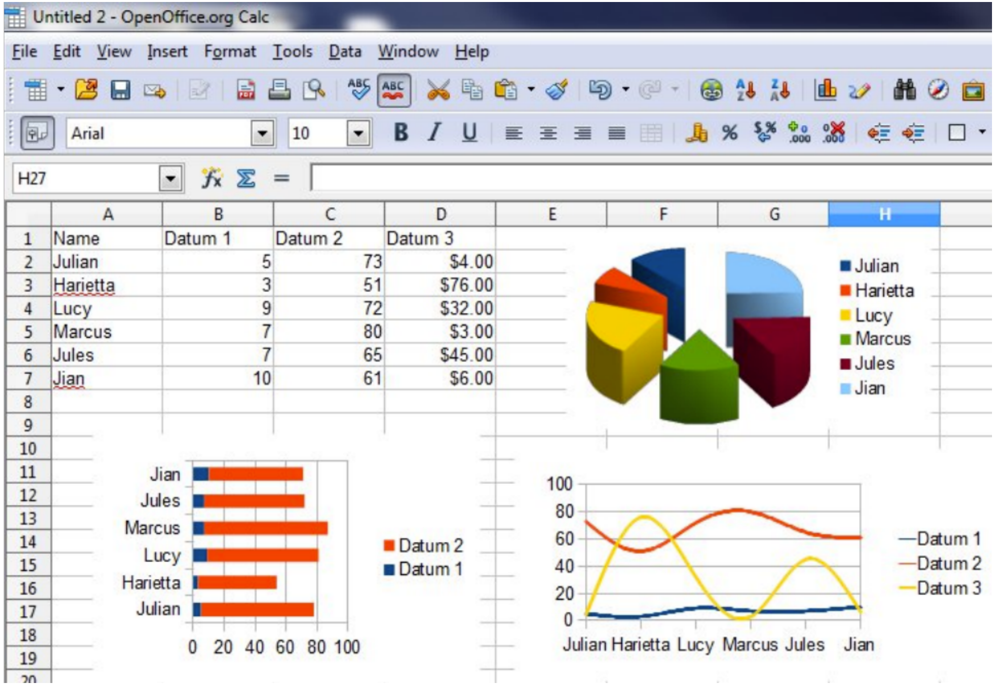
- State
- Strategy
- Observer
- Command
- Visitor

Behavioural Patterns: Observer

- We often have need to notify multiple subscribers about an event that occurs
  - We don't necessarily know which subscribers may be interested in our events
  - We might want to modify the subscriber list at run time

- Example: spreadsheet application with multiple graphs
  - Need to update graphs when spreadsheet data changes
  - Graphs can be added/removed at any time

Behavioural Patterns: Observer



Behavioural Patterns: Observer

Design Pattern:

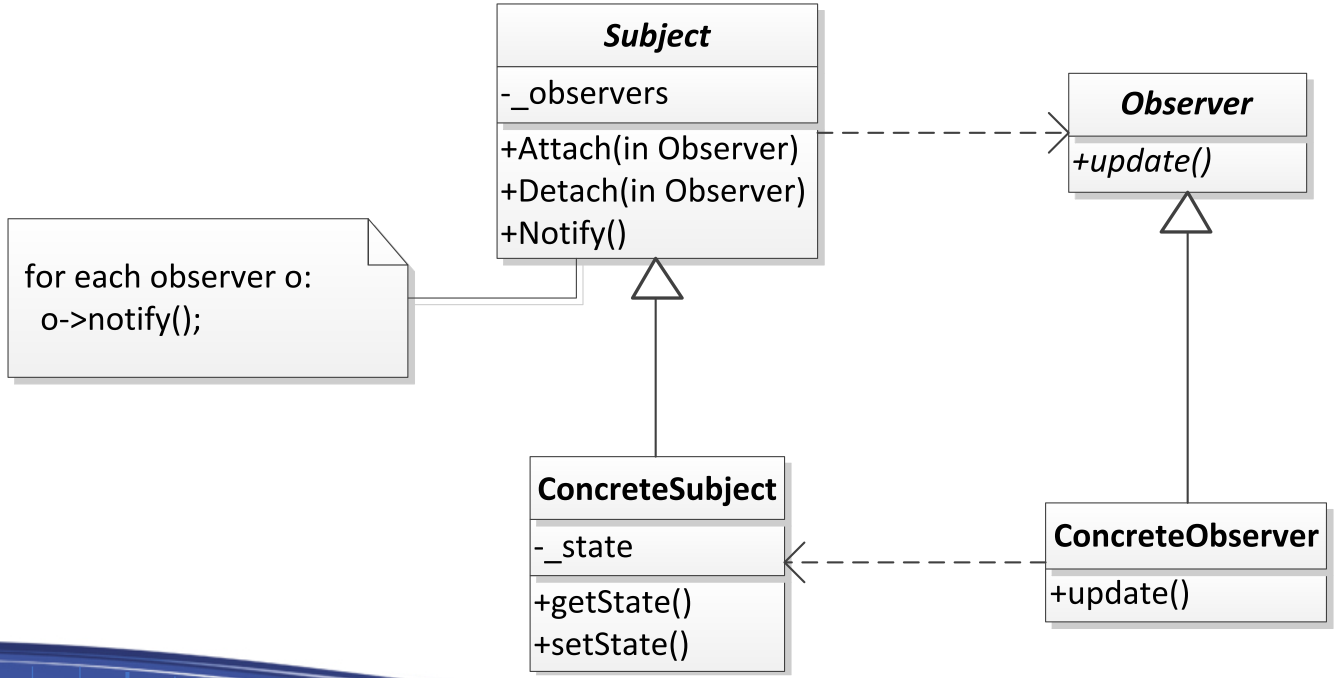
Observer

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Behavioural Patterns: Observer

- Applicability:
  - When an abstraction has two aspects, one dependent on the other; encapsulating these aspects in separate objects lets you vary and reuse them independently
  - When a change to one object requires changing others, and you don't know how many objects need to be changed
  - When an object should be able to notify other objects without making assumptions about who these objects are (i.e. we don't want these objects tightly coupled)

Behavioural Patterns: Observer



Behavioural Patterns: Observer

- Spreadsheet example:
  - Subject: Spreadsheet
  - ConcreteObserver: the various graphs
    - Pie, Bar, Line
    - When created, they are attached to the spreadsheet
  - Implementation of `update` is up to the individual graph classes
    - We could modify `update` to accept parameters passed from the spreadsheet
    - We could query the spreadsheet for the data we need

Behavioural Patterns: Observer

- Some Observers may observe more than one subject
  - We often pass in an event object containing details on which object generated the event as well as other pertinent information
- Deleting a Subject should cause Observers to remove any references to the Subject
  - Destructor of Subject can notify Observers of its deletion

Behavioural Patterns: Observer

- Consequences:
  - Abstract coupling between Subject and Observer; a Subject only knows that it has a list of Observers – it doesn’t know anything about them
  - Support for broadcast communication (one-to-many)
  - Unexpected updates
    - Observers have no knowledge of each other’s presence
    - A seemingly innocuous operation on a Subject may cause a cascade of updates to Observers and their dependent objects

Behavioural Design Patterns

- State
- Strategy
- Observer
- Command
- Visitor

Behavioural Patterns: Command

- Simple Example: Restaurant
  - Client of a restaurant wants to request food
  - Cook can only cook (execute) one food request at a time (and may currently be busy)
  - Waiter creates an abstraction of the request (the client's order) and places this request in the cook's queue

Behavioural Patterns: Command

- Suppose we are creating a set of classes to present a text-based menu system to a user

- We want to release the menu system as a library that can be integrated into other products

Behavioural Patterns: Command

Menu.h

```
class Menu
{
  public:

    virtual ~Menu();
    void add(MenuItem* item);
    MenuItem* getChoice();

  private:
    std::vector<MenuItem*> _items;

};
```

Behavioural Patterns: Command

- Problem: how do we design the `MenuItem` class?

- We don't know in advance what actions might be taken when a menu item is selected

- In other words, we need to issue requests to objects without knowing anything about the operation being requested, or the receiver of the request

Behavioural Patterns: Command

Design Pattern:

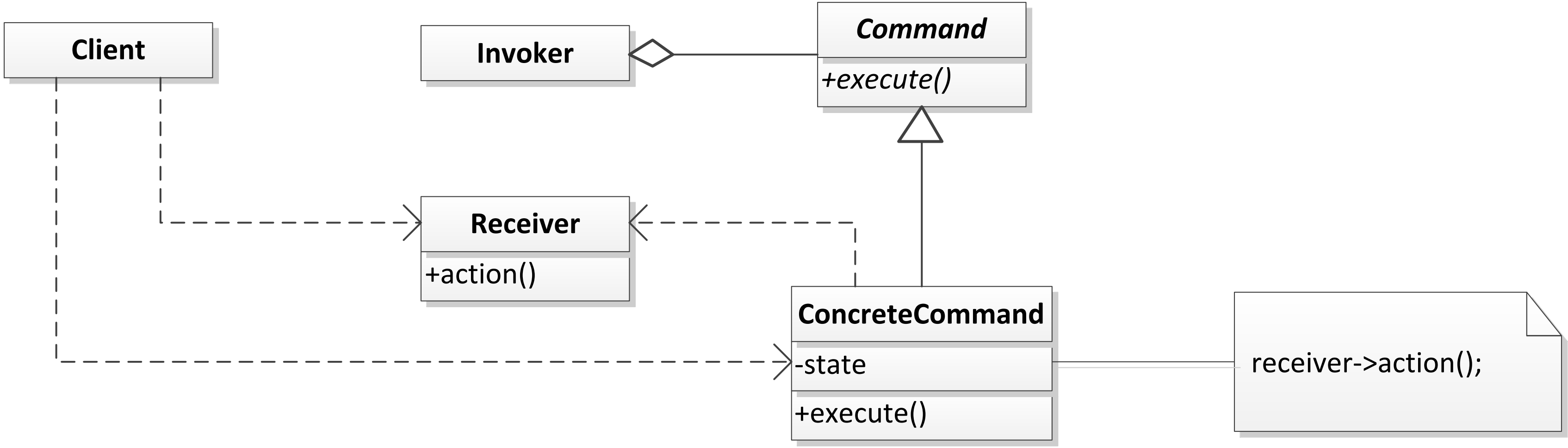
Command

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Behavioural Patterns: Command

- Applicability:
  - You want to parameterize objects by an action to perform
  - You want to specify, queue, and execute requests at different times
  - You want to support undo operations

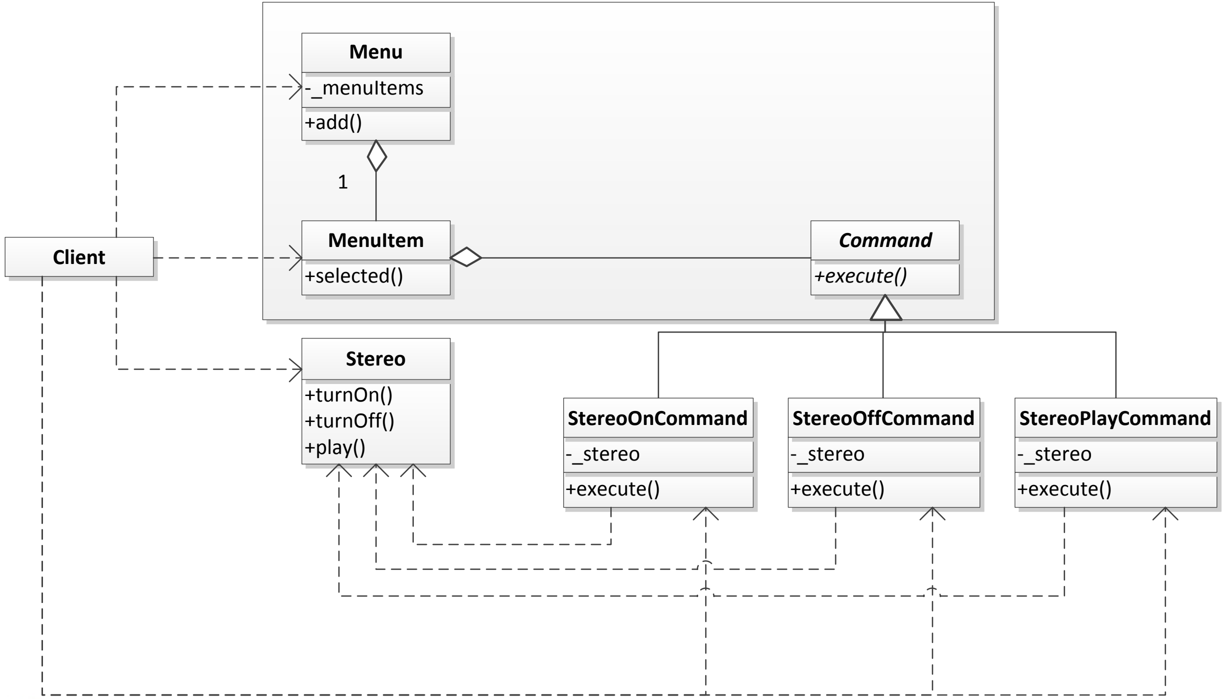
Behavioural Patterns: Command



Behavioural Patterns: Command

- Classes Involved:
  - Client
    - Creates the commands and sets the receiver object
    - Issues these commands to the invoker
  - Invoker
    - Maintains a queue (or stack, log) of commands
    - Execution of commands is done through invoker
  - Receiver
    - Implements any actions that may be executed by commands

Behavioural Patterns: Command



Behavioural Patterns: Command

main.cpp

```
Stereo* s = new Stereo("Living room stereo");

Command* cmd5 = new StereoOnCommand(s);
Command* cmd6 = new StereoOffCommand(s);
Command* cmd7 = new StereoPlayCommand(s);

Menu menu;

menu.add(new MenuItem("Turn on stereo", cmd5));
menu.add(new MenuItem("Turn off stereo", cmd6));
menu.add(new MenuItem("Play stereo", cmd7));

menu.add(new MenuItem("Turn on lamp", cmd1));
```

Behavioural Patterns: Command

- When a menu item is selected in the menu, the corresponding `MenuItem` is `selected()`

- It then invokes the `execute()` function from its associated `Command` object

- Requires dynamic dispatch

Behavioural Patterns: Command

Concrete Command – StereoOffCommand.h

```
    void execute()
    {
      this->_stereo->turnOff();
    }
```

Each `ConcreteCommand`, when `execute()`'d will implement their command by calling functions on the associated receiver

Behavioural Patterns: Command

- Consequences:
  - Decouples the object that invokes the operation from the one that knows how to perform it
  - Commands are first-class objects; they can be manipulated and extended like any other object
  - We can assemble commands into a composite command to allow for macro recording and playback
  - It is easy to add new Receivers and/or Commands, because we don't have to change existing classes
  - We can have multiple receivers

Behavioural Patterns: Command

Modified Client – main.cpp

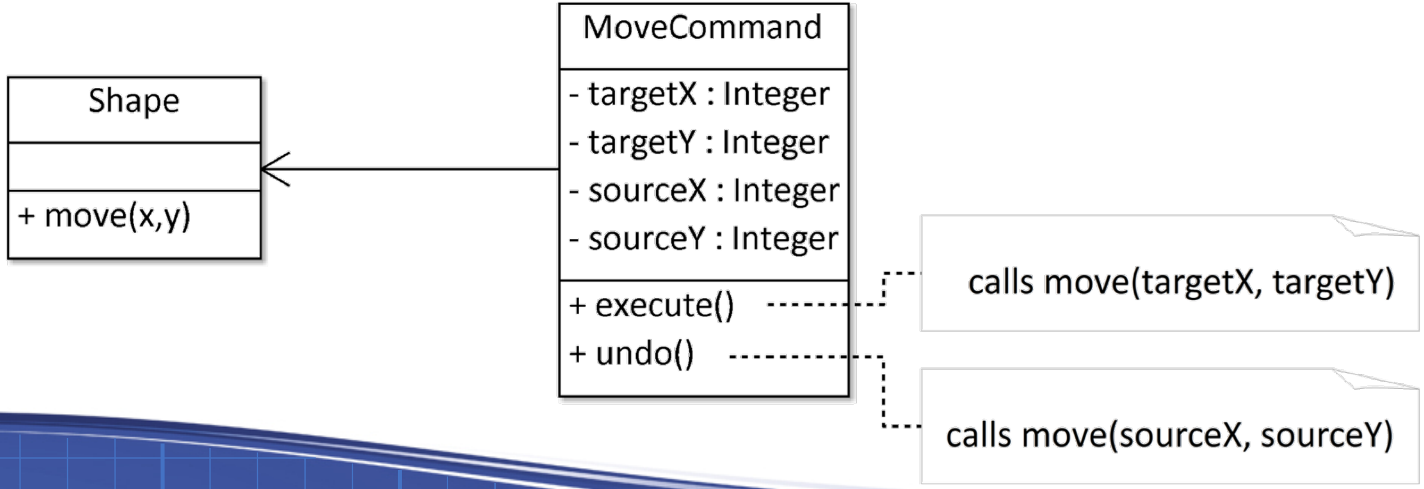
```
Light* l1 = new Light("Lamp");
Light* l2 = new Light("Porch light");

Command* cmd1 = new LightOnCommand(l1);
Command* cmd2 = new LightOffCommand(l1);
Command* cmd3 = new LightOnCommand(l2);
Command* cmd4 = new LightOffCommand(l2);
```

```
menu.add(new MenuItem("Turn off lamp", cmd2));
menu.add(new MenuItem("Turn on porch light", cmd3));
menu.add(new MenuItem("Turn off porch light", cmd4));
```

Behavioural Patterns: Command

- Consequences:
  - We can easily support rollback/undo operations by adding an `unexecute` or `undo` method



Behavioural Patterns: Command

- Consequences:
  - We can easily queue up commands to be executed as a batch
  - We can easily provide progress on a set of commands being executed

Behavioural Design Patterns

- State
- Strategy
- Observer
- Command
- Visitor

Behavioural Patterns: Visitor

- Suppose we have a hierarchy of employee classes:
  - `HourlyEmployee`, `SalariedEmployee`, etc.

- We need to be able to run reports on these employees. We may want:
  - A report of the earnings of all hourly employees
  - A report of the earnings of all employees
  - …

Behavioural Patterns: Visitor

- We don’t want to violate the Single Responsibility Principle by mixing reporting code into the employee classes

- We need to be able to add new reports at any given time
  - We don’t want to violate the Open-Closed Principle by having to modify the employee classes later

Behavioural Patterns: Visitor

Design Pattern:

Visitor

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

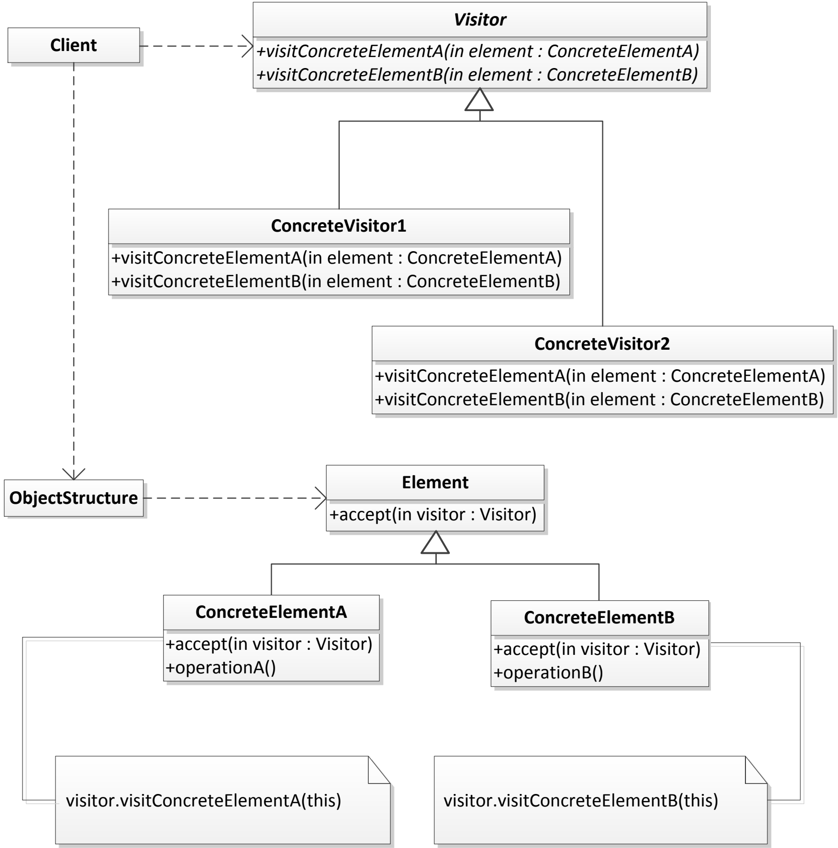
Behavioural Patterns: Visitor

- Applicability:
  - An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
  - Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid polluting their classes with these operations

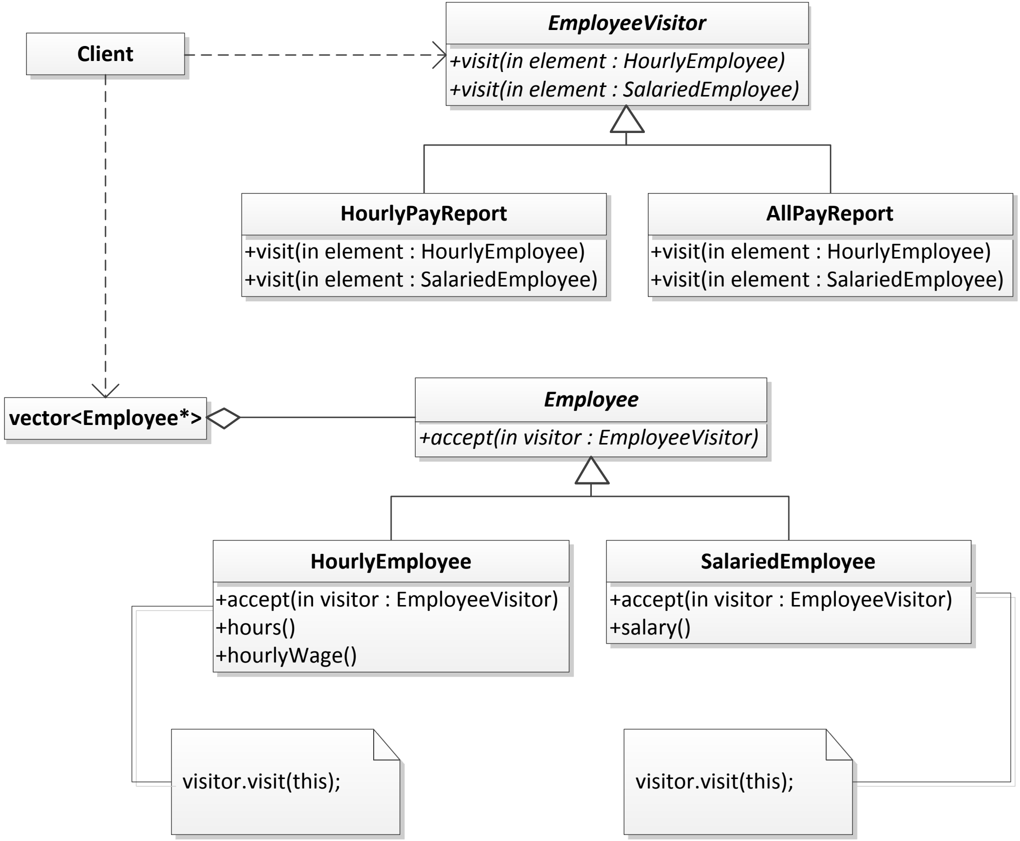
Behavioural Patterns: Visitor

- Applicability:
  - The classes defining the object structure rarely change (or cannot change), but you want to define new operations over the structure
  - For instance, we may be defining operations on third-party libraries classes to which we do not have the source code

Behavioural Patterns: Visitor



Behavioural Patterns: Visitor



Behavioural Patterns: Visitor

Employee.h

```
class Employee
{
  public:
    Employee(const std::string& name) : _name(name) { }

    const std::string name() const
    {
      return this->_name;
    }

    virtual void accept(EmployeeVisitor*) = 0;

  protected:
    std::string _name;
};
```

Behavioural Patterns: Visitor

HourlyEmployee.cpp

```
void HourlyEmployee::accept(EmployeeVisitor* visitor)
{
  visitor->visit(this);
}
```

SalariedEmployee.cpp

```
void SalariedEmployee::accept(EmployeeVisitor* visitor)
{
  visitor->visit(this);
}
```

Behavioural Patterns: Visitor

EmployeeVisitor.h

```
class EmployeeVisitor
{
  public:
    virtual void visit(HourlyEmployee*) = 0;
    virtual void visit(SalariedEmployee*) = 0;
};
```

Behavioural Patterns: Visitor

HourlyPayReport.h

```
class HourlyPayReport : public EmployeeVisitor
{
  public:
    HourlyPayReport(std::ostream&);
    virtual void visit(HourlyEmployee*);
    virtual void visit(SalariedEmployee*);

  private:
    std::ostream& _out;
};
```

Behavioural Patterns: Visitor

HourlyPayReport.cpp

```
void HourlyPayReport::visit(HourlyEmployee* e)
{
  this->_out << setw(20) << e->name();
  this->_out << setw(10) << e->hours();
  this->_out << "$" << setw(9) << e->hourlyWage();
  this->_out << "$" << (e->hours() * e->hourlyWage()) << endl;
}

void HourlyPayReport::visit(SalariedEmployee* e)
{
  // Do nothing
}
```

Behavioural Patterns: Visitor

AllPayReport.cpp

```
void AllPayReport::visit(HourlyEmployee* e)
{
  this->_out << setw(20) << e->name();
  this->_out << setw(10) << "n/a";
  this->_out << "$" << setw(9) << e->hourlyWage() << endl;
}

void AllPayReport::visit(SalariedEmployee* e)
{
  this->_out << setw(20) << e->name();
  this->_out << "$" << setw(9) << e->salary();
  this->_out << setw(10) << "n/a" << endl;
}
```

Behavioural Patterns: Visitor

main.cpp

```
main()
{
  vector<Employee*> employees;

  employees.push_back(new HourlyEmployee("Joe User", 60, 25.75));
  employees.push_back(new HourlyEmployee("Jane Doe", 55, 31.25));
  employees.push_back(new SalariedEmployee("Bob Caygeon", 75000));
  employees.push_back(new SalariedEmployee("Eve Adams", 72000));

  HourlyPayReport rpt1(cout);

  for (vector<Employee*>::iterator it = employees.begin(); it != employees.end(); ++it)
    (*it)->accept(&rpt1);    // Why not call rpt1.visit(*it)?  The visit() method requires a pointer
                             // to an instance of a concrete subclass, not the abstract parent class,
                             // as each concrete subclass is potentially treated differently.
  cout << endl;

  AllPayReport rpt2(cout);

  for (vector<Employee*>::iterator it = employees.begin(); it != employees.end(); ++it)
    (*it)->accept(&rpt2);
}
```

Behavioural Patterns: Visitor

Output

```
Hourly Employee Pay Report
Name                          Hours          Wages          Pay
=======================================================================
Joe User                      60             $25.75  $1545
Jane Doe                      55             $31.25  $1718.75
Employee Pay Report
Name                          Salary         Wage
=======================================================================
Joe User                      n/a            $25.75
Jane Doe                      n/a            $31.25
Bob Caygeon                   $75000  n/a
Eve Adams                     $72000  n/a
```

Behavioural Patterns: Visitor

- Consequences:
  - Visitor makes adding new operations easy
  - A visitor gathers related operations and separated unrelated ones
  - Accumulating state
  - Adding new ConcreteElement classes is hard

Behavioural Design Patterns

- State
- Strategy
- Observer
- Command
- Visitor