Copyright Warning

ViSLAB
High-Performance Visualization, Communication, Computing

Object Oriented Design

Week 11-1

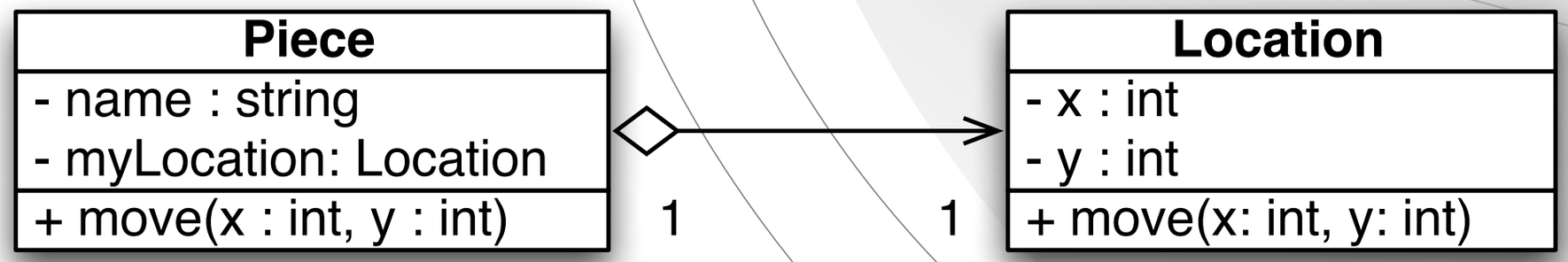
- Generation :
  - Prototype
- Structural:
  - Facade
  - Proxy
- Behaviour:
  - State

Prototype

- One of "Generation" pattern
- Hides details of object generation.
  - have an object ready for "copy"

- Example:
  - Design Chess
    - Piece
    - Location

Prototype



- Piece
  - name
  - location
    - move() : calls Location's move().
- Location
  - x, y
  - move()

before D.P

```
class Piece {
    string name;
    Location* myLocation;
public:
    Piece() : myLocation(NULL) {}
    ~Piece() { if (myLocation!=NULL) delete myLocation}
    string getName() const {return name;}
    void setName(string name) {this->name = name;}
    Location *getLocation() const {return myLocation;}
    void setLocation(Location* loc) {this->myLocation = loc;}
    void move(int x, int y) {myLocation->move(x, y);}
};

class Location {
    int x, y;
public:
    Location(int x, int y) : x(x), y(y) {}
    int getX() const {return x;}
    int getY() const {return y;}
    void move(int x, int y) {this->x += x; this->y += y;}
};
```

before D.P

```
int main() {
Piece* pPb1 = new Piece();
pPb1->setName("Pawn-black");
pPb1->setLocation(new Location(8, 7));
pPb1->move(0, -1);

Piece* pPb2 = new Piece();
pPb2->setName("Pawn-black");
pPb2->setLocation(new Location(7, 7));
pPb2->move(0, -2);
...
}
```

- Piece and Location need to be “new”-ed.
- can we copy what we already have?

after Prototype

- “Cloneable” interface
- can we copy what we already have?

```
class Cloneable {
public:
   virtual Clonable* clone() = 0;
};
```

after Prototype

- "Cloneable" interface

```
public Location : public Cloneable {
public:
    ...
    Cloneable *clone() {return new Location(x, y);}
    ...
}

class Piece : public Cloneable {
public:
    ...
    Cloneable* clone() {
        Piece* clone = new Piece();
        clone->setLocation((Loation*) myLocation->clone());
        return clone;
    }
};
```

after Prototype

```
int main() {
    Piece* pPb1 = new Piece();
    pPb1->setName("Pawn-black");
    pPb1->setLocation(new Location(8, 7));
    pPb1->move(0, -1);

    Piece* pPb2 = new Piece();
    pPb2->setName("Pawn-black");
    pPb2->setLocation(new Location(7, 7));
    pPb2->move(0, -2);
    ...
}
```

```
int main() {
    Piece* pPb1 = new Piece();
    pPb1->setName("Pawn-black");
    pPb1->setLocation(new Location(8, 7));
    pPb1->move(0, -1);

    Piece* pPb2 = (Piece*)pPb1->clone();
    pPb2->move(-1, 0);
    ...
}
```

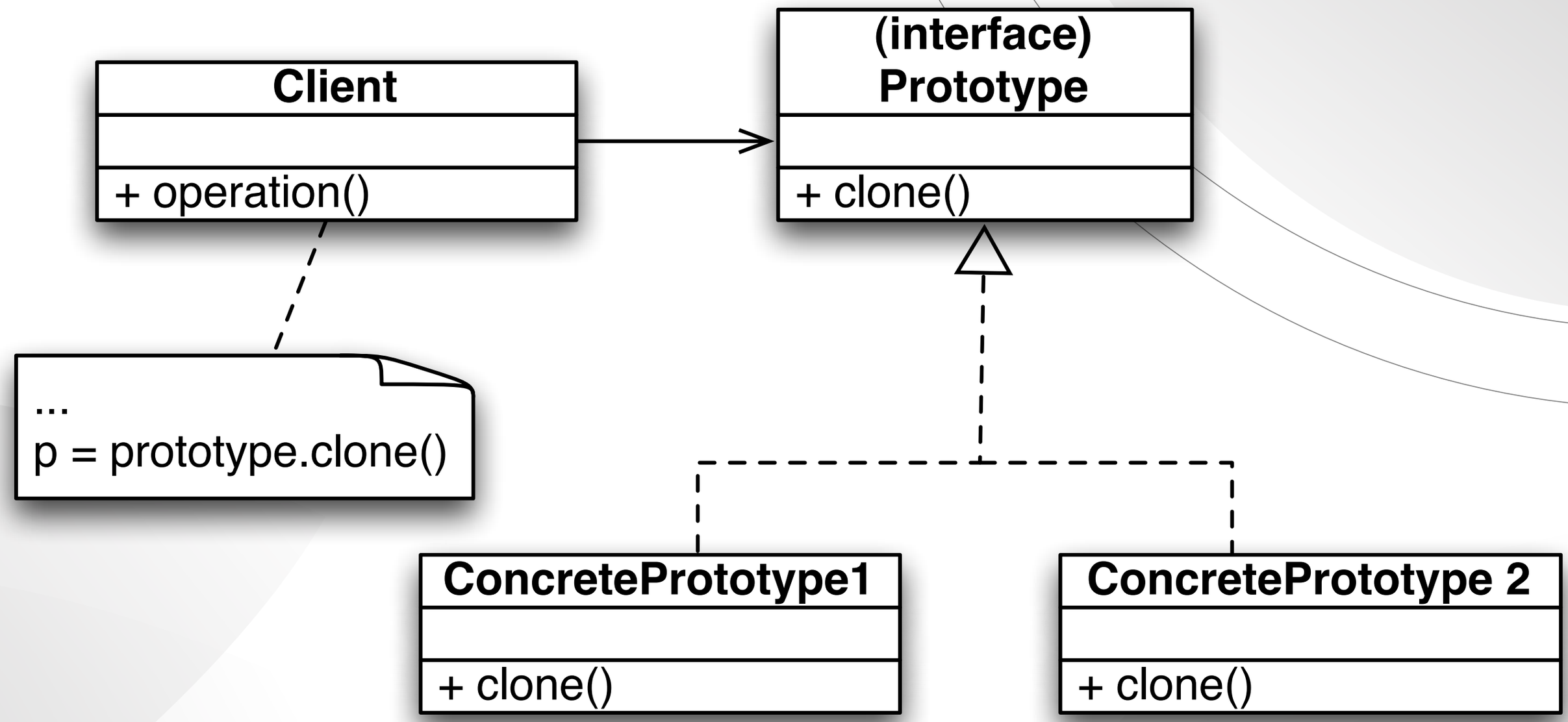
Prototype

- Merit
  - simplify the object creation process
    - you need have a prototype.
  - dynamically increase the types
    - no need for making subclasses
  - dynamically change system behaviours
    - by changing prototype

Prototype

- be careful...
- shallow or deep copy?
- infinite loop (A contains B, B contains A)

Prototype



Facade

- Provide common interfaces
- Prevents complex dependencies
  - by providing a single entry point

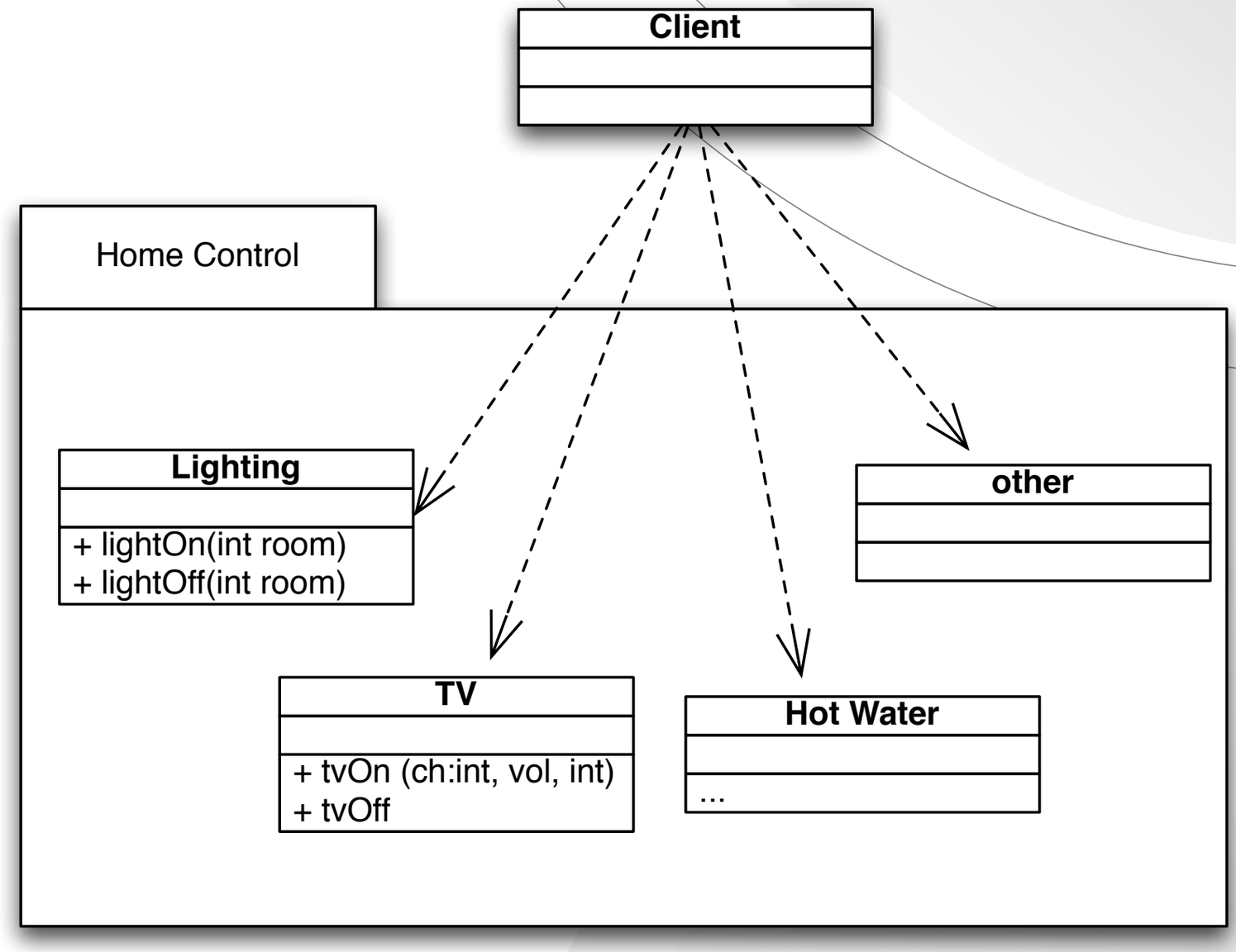
- Example:
  - Home Control System
    - room lights, TV, air-cond., hot water, etc.
    - control one device at the time

before D.P

```
class Lighting {
public:
    void lightOn(int room) {
        if (room == 0) cout << "room0's light is on." << endl;
        else if (room == 1) cout << "room1's light is on" << endl;
        else cout << "no op." << endl;
    }
    void lightOff(int room) {
        if (room == 0) cout << "room0's light is off." << endl;
        else if (room == 1) cout << "room1's light is off" << endl;
        else cout << "no op." << endl;
    }
};

class TV {
public:
    void tvOnOn(int ch, int vol) {
        cout << "TV is on." << endl;
        cout << "Channel is " << ch << endl;
        cout << "Volume is " << vol << endl;
    }
    void tvOff() {
        cout << "TV is off." << endl;
    }
};
```

before D.P

```
int main() {
...
Lighting *lighting = new Lighting();
TV* tv = new TV();
...
lighting->lightOn(0);
tv->tvOn(2, 5);
...
}
```



before Facade

- client needs to know how to interface each device
- client needs to know the interface of a new device

after Facade

- HomeControlFacade
  - only class/object external clients need to interface

```
class HomeControlFacade {
public:
    void homeControlPreset() {
        Lighting* light = new Lighting();
        TV* tv = new TV();
        ...

        light->lightOn(3);
        tv->tvOn(2, 5);
        ...
    }
};
```

Facade

- Merit
  - simplify the interface to the sub-systems
  - changes in sub-systems do not affect the interface
  - client only needs to understand the facade class
  - encapsulate the sub-systems

Facade

- be careful...
  - remove fine controls (for sub-systems)

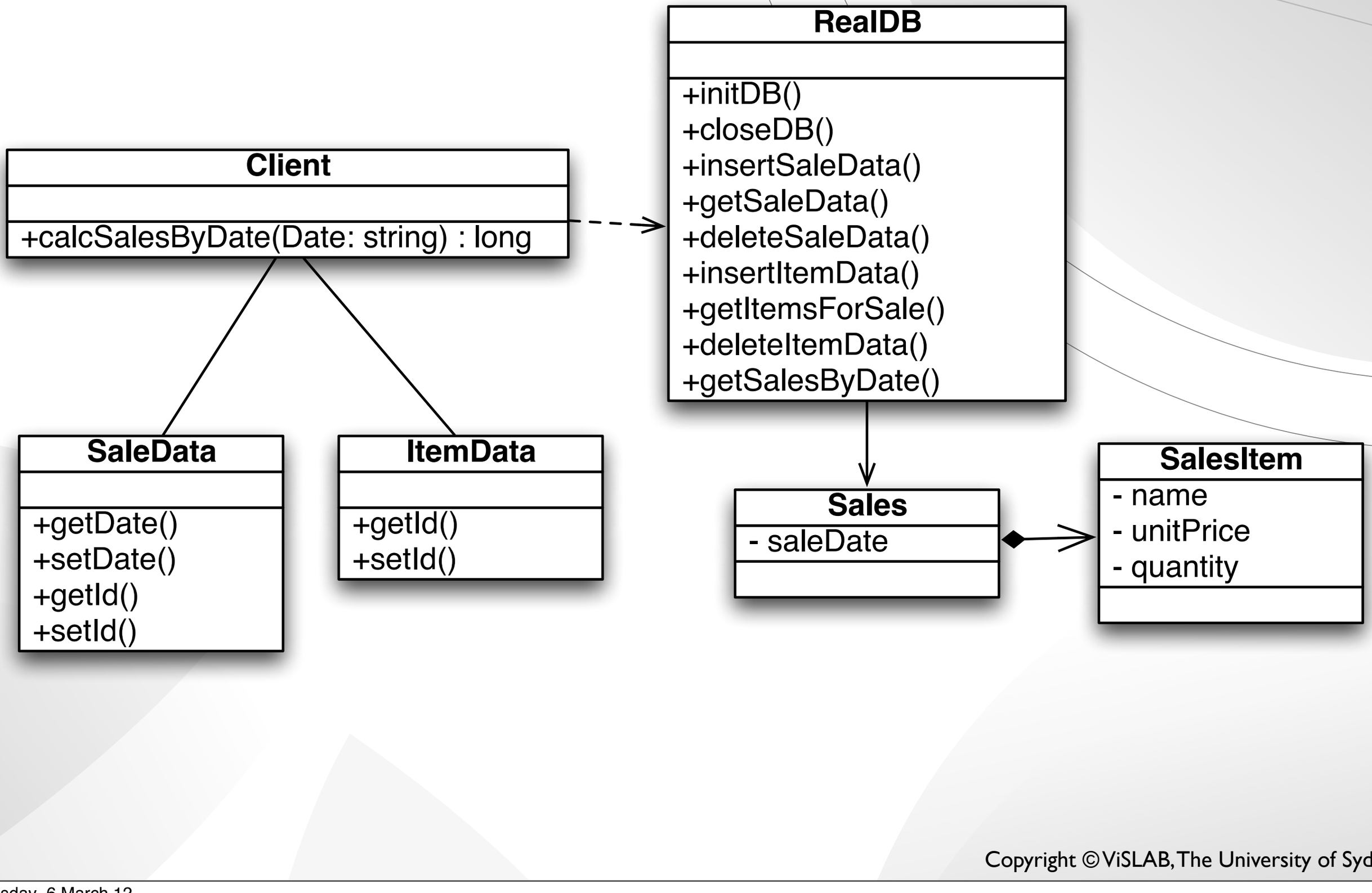
Proxy

- Provide mechanism to separate two different logics
  - HTML/display and database access
  - Web service and RMI/CORBA processes

- Example:
  - A System retrieve data from a database
    - database is independent module
    - proxy to interface the database

```
class ItemData {
    int id;
    int saleId;
    string name;
    int unitPrice;
    int quantity;
public:
    int getId();
    void setId(int id);
    string getName();
    void setName(string name);
    int getUnitPrice();
    void setUnitPrice(int unitPrice);
    int getQuantity();
    void setQuantity(int quantity);
};

class SaleData {
    int id;
    string date;
public:
    string getDate();
    int getId();
    void setDate(string date);
    void setId(int id);
};
```

```
class RealDB {
public:
    static void initDB();
    static void closeDB();
    static void insertSaleData(SaleData* sale);
    static SaleData* getSaleData(int id);
    static void deleteSaleData(SaleData* sale);
    static void insertItemData(ItemData* sale);
    static list<ItemData*>* getItemsForSale(int id);
    static void deleteItemData(ItemData* sale);
    static list<SaleData*>* getSalesByDate(string data);
};

long calcSalesByDate(string date) {
    long total = 0;
    list<SaleData*>* sales = RealDB::getSalesByDate(data);
    list<SaleData*>::iterator it = sales->begin();
    while (it != sales->end()) {
       SaleData* sale = *it++;

       list<ItemData*>* items = RealDB::getItemForSale(sale->getId());
       list<ItemData*>::iterator itI = items->begin();
       while (itI != items->end()) {
          ItemData* item = *itI++;
          total += item->getUnitPrice() * item->getQuantity();
       }
    }
    return total;
}
```



after Proxy

- previous example requires working RealDB.

```
class DataBase {
public:
    virtual void initDB() = 0;
    virtual void closeDB() = 0;
    virtual void insertSaleData(SaleData* sale) = 0;
    virtual SaleData* getSaleData(int id) = 0;
    virtual void deleteSaleData(SaleData* sale) = 0;
    virtual void insertItemData(ItemData* sale) = 0;
    virtual list<ItemData*>* getItemsForSale(int id) = 0;
    virtual void deleteItemData(ItemData* sale) = 0;
    virtual list<SaleData*>* getSalesByDate(string data) = 0;
};
```

after Proxy

- use DataBase interface

```
long calcSalesByDate(DataBase* db, string date) {
    long total = 0;

    list<SaleData*>* sales = db::getSalesByDate(data);
    list<SaleData*>::iterator it = sales->begin();
    while (it != sales->end()) {
       SaleData* sale = *it++;


       list<ItemData*>* items = db::getItemForSale(sale->getId());
       list<ItemData*>::iterator itI = items->begin();
       while (itI != items->end()) {
          ItemData* item = *itI++;
          total += item->getUnitPrice() * item->getQuantity();
       }
    }
    return total;
}
```

after Proxy

- use DataBase interface

```
class ProxyDB : public DataBase {
    bool real;
    DataBase* db;
    bool isReal() {return real;}
public:
    ProxyDB() : real(false) {
        db = (isReal()) ? new RealDB() : MockDB();
    }
    ~ProxyDB() {delete db;}
    void useRealDB(bool b) {real = b;}
    virtual void initDB() {db->initDB();}
    virtual void closeDB() {db->closeDB();}
    virtual void insertSaleData(SaleData* sale) {
        db->insertSaleData(sale);
    }
    virtual SaleData* getSaleData(int id) {
        return db->getSaleData(id);
    }
    ...
};
```

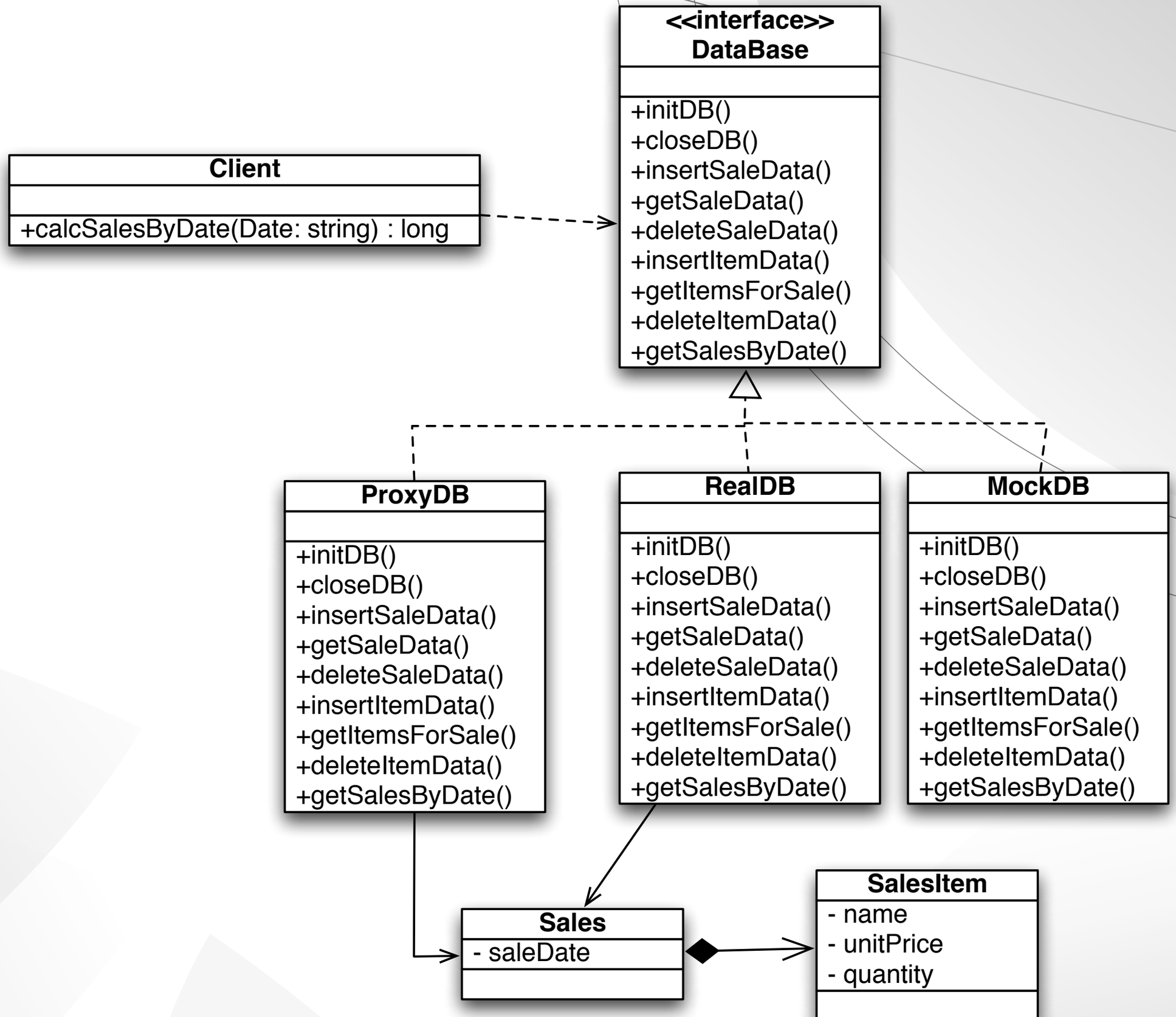
after Proxy

```
class MockDB : public DataBase {
    list<SaleData*>* sales;
    list<ItemData*>* items;
public:
    virtual void initDB() {
        sales = new list<SaleData*>;
        items = new list<ItemData*>;
    }
    virtual void closeDB() {
        delete sales;
        delete items;
    }
    virtual void insertSaleData(SaleData* sale) {
        sales->push_back(sale);
    }
    virtual SaleData* getSaleData(int id) {
        list<SaleData*>::iterator it = sales->begin();
        while(it != sales->end()) {
            SaleData* sale = *it++;
            if (sale->getId() == id) return sale;
        }
        return NULL;
    }
    ...
};
```

afterProxy

- change in the client code:
- DataBase* db = new RealDB();

- DataBase* db = new ProxyDB();



Proxy

- Merit
  - can hide the location of external resources
  - clear interface
  - can have a clear place for performance improvement.

State

- Needs to change behaviours based on
  - object's internal state
- when an object has many different states
  - condition statements might get complex
- a class represents one behaviour for a state

State

- Excample
  - Engine
    - states: Idle, Low, and High
    - Up or Down changes its state
  - client
    - create an engine
    - up() and down() to change engine's state

State

```
int main() {
Engine* mEng = new Engine();
mEng->up();
... // switch between up() and down()
mEng->down();
    mEng->state->showCurentState();
    ...
    return 0;
}
```

before D.P

```
    class Engine {
    void currentState();
    int state;
public:
    Engine() : state(0) {}
    void up();
    void down();
};

void Engine::up() {
    if      (state == 0) state = 1; // Idle -> low
    else if (state == 1) state = 2; // low -> high
    currentState();
}

void Engine::down() {
    if      (state == 2) state = 1; // High->low
    else if (state == 1) state = 0; // low -> Idle
    currentState();
}

void Engine::currentState() {
    if      (state == 0) cout << "State:Idle" << endl;
    else if (state == 1) cout << "State:Low" << endl;
    else if (state == 2) cout << "State:High" << endl;
}
```

State

- there might be more states in the future
- treat each state as a class
- create a general interface

```
class EngineState {
public:
   virtual void up(Engine* eng) = 0;
   virtual void down(Engine* eng) = 0;
   virtual void currentState() = 0;
};
```

```
    class EngineIdle : public EngineState
    {
public:
    void up(Engine* eng) {
        cout << "Idle->Low" << end;
        eng->changeState(new EngineLow());
    }
    void down(Engine* eng) {
        cout << "No Change" << endl;
        eng->changeState(new EngineIdle());
    }
    void currentState() {
        cout << "State:Idle" << endl;
    }
};
```

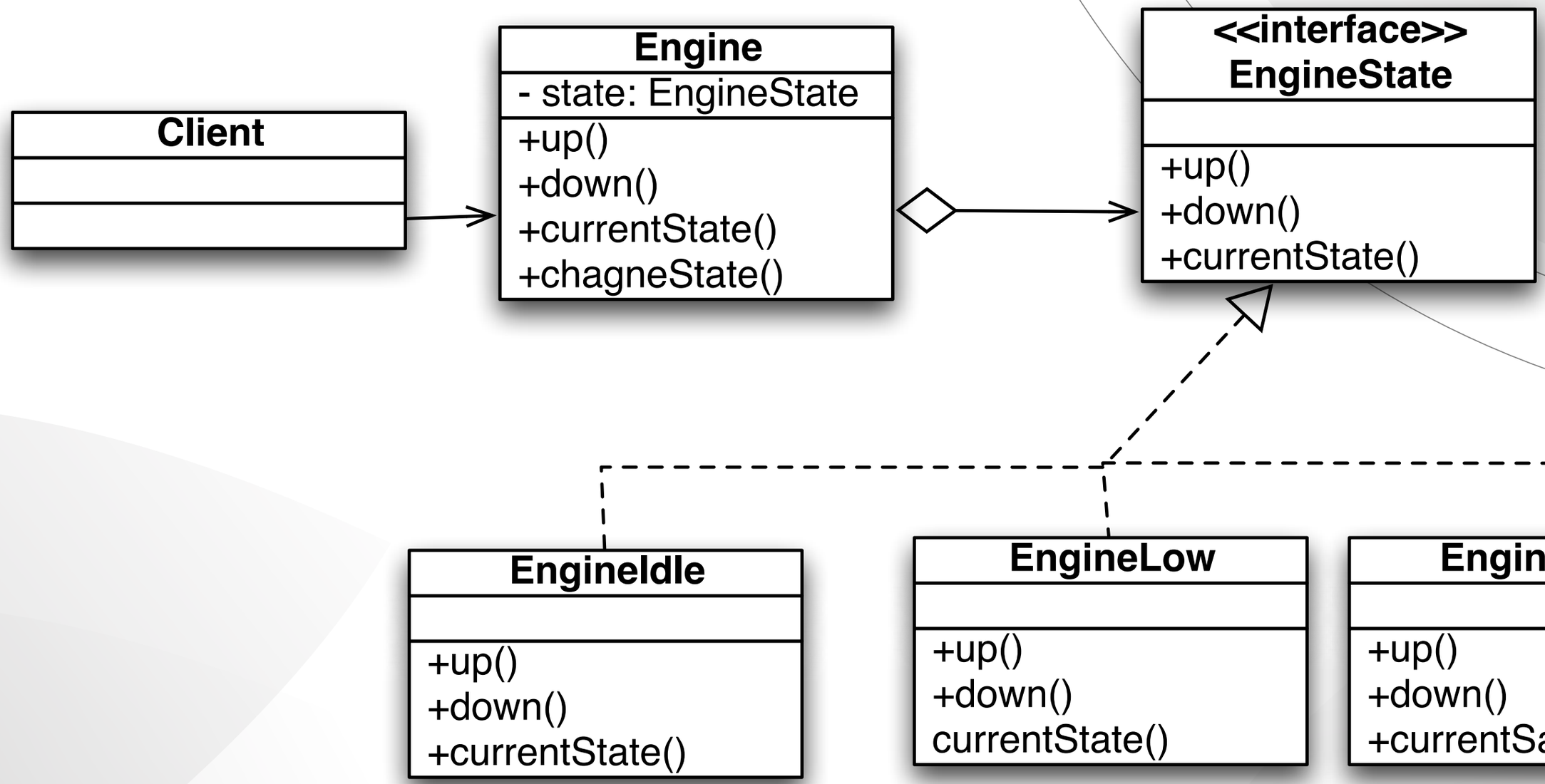
```
class EngineLow : public EngineState {
public:
    void up(Engine* eng) {
        cout << "Low->High" << end;
        eng->changeState(new EngineHigh  ));
    }
    void down(Engine* eng) {
        cout << "Low->Idle" << endl;
        eng->changeState(new EngineIdle());
    }
    void currentState() {
        cout << "State:Low" << endl;
    }
};

class EngineHigh : public EngineState {
public:
    void up(Engine* eng) {
        cout << "No Change" << end;
        eng->changeState(new EngineHigh());
    }
    void down(Engine* eng) {
        cout << "High->Low" << endl;
        eng->changeState(new EngineLow());
    }
    void currentState() {
        cout << "State:High" << endl;
    }
};
```

State

- Engine holds EngineState
- actual processing is done by State object.

```
class Engine {
    EngineState* state;
public:
    Engine() : state(new EngineIdle()) {}
    ~Engine() {delete state;}
    void up() {state->up(this);}
    void down() {state->down(this);}
    void changeState(EngineState* newState) {
        delete state;
        state = newState;
    }
    void currentState() {
        state->currentState();
    }
};
```

State



State

- Merit
  - No overlaps of logics

- careful...more classes (high initial investment)