Aggregation

- Construction of an object from others
- Object may have sub-objects contained within it

Advantages

- Simple – deal with the containing object rather than all of them
- Safe – sub-objects encapsulated
- Specialized interface – general objects used together with interface specific to problem
- Structure indicates designers intention
- Can substitute implementations

Types of Aggregation

- Static – number of sub-objects cannot vary
  - A Frame has a Location and Shape
  - A Customer has a Name and an Address
  - Dynamic – number of sub-objects may vary
  - A catalog has many catalog items
  - A phone list has changing entries

Aggregation in Diagram

Static

Dynamic
Aside: Encapsulation

- Encapsulation – preventing access to internal data of object
- Always want to consider how much user of class needs to know about its internals
  - Less is better for design flexibility
  - More can be better for efficiency
- Our focus always on “good” design

Ex. Static Aggregation (1)

Counter for bus passengers by payment method

Passenger Counter Class

class Counter {
    public:
        Counter() : cnt(0) {};
        Counter(const Counter& c) : cnt(c.cnt) {};
        Counter(int c) : cnt(c) {};
        void increment() { cnt++ ; };
        int getCount() { return cnt; };
    private:
        int cnt;
};

Passenger Counter Class

class PassengerCounter {
    public:
        PassengerCounter();
        PassengerCounter(const PassengerCounter&);
        void incUniv();
        void incMonthly();
        void incCash();
        int getUnivCount() const;
        int getMonthlyCount() const;
        int getCashCount() const;
    private:
        PassengerCounter univ, monthly, cash;
};

PassengerCounter Constructors

PassengerCounter::PassengerCounter() :
    univ(), monthly(), cash() {};

PassengerCounter::PassengerCounter(const PassengerCounter& p) :
    univ(p.univ), monthly(p.monthly),
    cash(p.cash) {};

PassengerCounter Mutators

void PassengerCounter::incUniv() {
    univ.increment();
};

void PassengerCounter::incMonthly() {
    monthly.increment();
};

void PassengerCounter::incCash() {
    cash.increment();
};
PassengerCounter Accessors

```
int PassengerCounter::getUnivCount() const {
    return univ.getCount();
}
int PassengerCounter::getMonthlyCount() const {
    return monthly.getCount();
}
int PassengerCounter::getCashCount() const {
    return cash.getCount();
}
```

Aside: Equality

- When are two Counters equal?
  - When are same object?
  - When have same value?
- Which is more appropriate depends on class
- Provide equality predicate or operator for class

Equality: Same Object

Check addresses of objects

```
bool Counter::equalTo(const Counter& c) {
    return (this == &c);
}
```

Silly unless have pointers to Counter objects

Equality: Same Value

```
bool Counter::operator==(const Counter& c) {
    return (this == &c);
}
```

Ex. Static Aggregation (2)

```
Point Class

class Point {
    public:
    Point() : x(0), y(0) {}  
    Point(const Point& p) : x(p.x), y(p.y) {}  
    Point(int x_, int y_) : x(x_), y(y_) {}  
    int getX() { return x; }  
    int getY() { return y; }  
    Point translate(int x_delta, int y_delta) {
        return Point(x+x_delta, y+y_delta);  
    }
    private:
    int x, y;
};
```
Dimension Class

class Dimension {
public:
    Dimension() : width(0), height(0) {}  
    Dimension(const Dimension& d) : 
        width(d.width), height(d.height) {}  
    Dimension(int w, int h) : width(w), height(h) {}  
    int getWidth() const { return width; }  
    int getHeight() const { return height; }  
    // possibly some other methods  
private:  
    int width, height; 
};

Rectangle Class

class Rectangle {
public:
    Rectangle (Point corner, Dimension shape);  
    void MoveUp (int deltaY);  
    void MoveDown (int deltaY);  
    void MoveLeft (int deltaX);  
    void MoveRight (int deltaX);  
    void Draw (Canvas & canvas);  
    void Clear (Canvas & canvas);  
    ~Rectangle ();  
private:  
    Point upperLeft;  
    Dimension area;  
};

Rectangle Implementation (1)

Rectangle::Rectangle(Point corner, Dimension shape) : 
upperLeft(corner), area(shape) {}  

void Rectangle::MoveUp(int deltaY) {  
    upperLeft = upperLeft.translate(0, deltaY);  
}  // ... MoveDown, MoveLeft, MoveRight similar to MoveUp

Rectangle Implementation (2)

void Rectangle::Draw(Canvas & canvas) {  
    Point up_rgt = upperLeft.translate(area.getWidth(), 0);  
    Point lw_lft = upperLeft.translate(0, area.getHeight());  
    canvas.DrawLine(upperLeft, up_rgt);  
    canvas.DrawLine(up_rgt, lw_lft);  
    canvas.DrawLine(lw_lft, upperLeft);  
}

Dynamic Aggregation

- Object with variable numbers of sub-objects  
- Example: polygonal figure with arbitrary number of sides
### PolyShape Aggregation

- **PolyShape**
- **Point**

### Detailed PolyShape Aggregation

- **PolyShape**
- **PointNode**
- **Point**

### PolyShape Structure

- **PolyShape**
- **Object**
- **head**
- **tail**
- **PointNode**
- **next**
- **Point**

### PolyShape Class

```cpp
class PolyShape {
public:
    PolyShape(int x, int y);
    void Add(const Point& newpoint);
    void Draw(Canvas& canvas);
    ~PolyShape();
private:
    PointNode *head;
    PointNode *tail;
    int length;
};
```

### PolyShape Implementation

```cpp
PolyShape::PolyShape (int x, int y) :
    head(0), tail(0), length(1)
{
    head = tail = new PointNode(Point(x, y));
}
```

### Adding Point to PolyShape

```cpp
// Add a point to polygon
void PolyShape::Add (const Point& newpoint) {
    PointNode *newNode = 
    new PointNode(newpoint);
    tail->Next(newNode);
    tail = newNode;
    length = length + 1;
}
```
Drawing A PolyShape

```cpp
void PolyShape::Draw (Canvas& canvas) {
    if (length == 1) return;
    PointNode *node, *next;
    node = head;
    while (node != tail)
        next = node->Next();
        canvas.DrawLine(node->Contents(), next->Contents());
        node = next;
    canvas.DrawLine(head->Contents(), tail->Contents());
}
```

PointNode Class

```cpp
class PointNode {
public:
    PointNode(const Point& loc);
    PointNode* Next();
    void Next(PointNode* nst);
    Point& Contents();
    ~PointNode();
private:
    PointNode* next;
    Point       *location;
};
```

PointNode Implementation

```cpp
PointNode::PointNode(const Point& loc) {
    location = new Point(loc); // need location copy
    next = (PointNode*)0;
}
PointNode* PointNode::Next() { return next; }
void PointNode::Next(PointNode* nst) { next = nst; }
Point& PointNode::Contents() { return *location; }
PointNode::~PointNode() { delete location; }
```

PolyShape Destructor

```cpp
PolyShape::~PolyShape() {
    PointNode *next = head;
    while (next) {
        PointNode *node = next->Next();
        delete next;
        next = node;
    }
}
```