Object-Oriented Programming

Encapsulation

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Credits

Adapted from: Encapsulation is not information hiding

Definitions

- Encapsulation

  The bundling of data with the methods that operate on that data

- Information Hiding

  Hide the internal representation, or state, of an object from the outside

- Note: Some definitions of *encapsulation* include the definition of *information hiding*

Language Features

- C language support:

  `struct` with fields only and no access specifiers

  free functions

- C++ language support:

  `class` Integrate data with methods

  Access specifiers: `public`, `private`, `protected`

Access & Visibility for Class *A*

Access Type	Visibility
`public`	All code
`protected`	`class A` methods friend functions of `class A` class derived from `class A`
`private`	`class A` methods friend functions s of `class A`

First Rule: Limit the Interface

- Limit the methods available
- Limit access and visibility

  I.e., `private` whenever possible

- Remove any unneeded parameters

  E.g., code where a local variable could be used instead of a parameter

- Use `const` whenever possible for reference and pointer parameters and methods

Minimum Essential Interface

- Unless required, more is **not** better

- Developers tend to overdesign and provide more than is needed (sometimes missing what is needed)

  E.g., A study at Microsoft showed that 30% of methods identified in the design of software were never implemented

- Start with the minimum functionality and configurability, and add as needed

Position v0

```
struct Position {
    double latitude;
    double longitude;
};

// distance from first to second position
double distance(const Position& position1,
                const Position& position2);

// heading from first to second position
double heading(const Position& position1,
               const Position& position2);
```

```
// position of office
Position office;
office.latitude = 36.538611;
office.longitude = -121.797500;

// position of local coffee shop
Position coffeeShop;
coffeeShop.latitude = 36.539722;
coffeeShop.longitude = -121.907222;

// calculate distance and heading from office to coffee shop
auto coffeeDistance = distance(office, coffeeShop);
auto coffeeHeading  = heading(office, coffeeShop);
```

Position v0 Analysis

```
struct Position {
    double latitude;
    double longitude;
};

// distance from first to second position
double distance(const Position& position1,
                const Position& position2);

// heading from first to second position
double heading(const Position& position1,
               const Position& position2);
```

- Data, `latitude` and `longitude`, in the Position class

- Operations, `distance()` and `heading()`, are free functions in a separate file

- The client program has to include two files

- Easy to get the order of arguments backward `heading()` and calculate the opposite of what is needed

Other Position v0 Issues

```
struct Position {
    double latitude;
    double longitude;
};

// distance from first to second position
double distance(const Position& position1,
                const Position& position2);

// heading from first to second position
double heading(const Position& position1,
               const Position& position2);
```

- Fields of struct `Position` are public and directly accessible. It is possible to store invalid latitude and longitude values, producing garbage distance and headings (GIGO)
- Changes to the field/data member names, types, or even storage in `Position` require changes in the implementation of `distance()` and `heading()` (Δ Position.hpp ⟶ Δ PathCalculations.cpp)

Encapsulation Rule

Place data and the operations that perform on that data in the same class

- Don't make the client tie data and operations together

- Provide it in one class (or the fewest number of classes needed)

- Improves class cohesion

Position v1

```
class Position {
public:
    double latitude;
    double longitude;

    // distance to a position
    double distance(const Position& position);

    // heading to a position
    double heading(const Position& position);
};
```

```
// position of office
Position office;
office.latitude = 36.538611;
office.longitude = -121.797500;

// position of local coffee shop
Position coffeeShop;
coffeeShop.latitude = 36.539722;
coffeeShop.longitude = -121.907222;

// calculate distance and heading from office to coffee shop
auto coffeeDistance = office.distance(coffeeShop);
auto coffeeHeading  = office.heading(coffeeShop);
```

Position v1 Analysis

```
class Position {
public:
    double latitude;
    double longitude;

    // distance to a position
    double distance(const Position& position);

    // heading to a position
    double heading(const Position& position);
};
```

- Only need to include a single include file, `Position.hpp`
- Clear direction of operations `heading()`

Information Hiding Rules

1. Don't expose data items

2. Don't expose the difference between stored data and derived data

3. Don't expose the implementation details of a class

4. Don't expose a class's internal structure

Information Hiding Rule 1

Don't expose data items

- Make all data members private

- If access is needed externally to the class, use get and set methods for access

- Isolates client from changes to data members (if the same external functionality is needed)

Position v2

```
class Position {
public:

    /* Set position

       @param latitude Requires -90 <= latitude <= 90
       @param longitude Requires -180 < longitude <= 180
       @return if position is legitimate
    */
    bool set(double latitude, double longitude);

    // latitude
    double getLatitude() const;

    // longitude
    double getLongitude() const;

    // distance to a position
    double distance(const Position& position);

    // heading to a position
    double heading(const Position& position);

private:
    double latitude;
    double longitude;
};
```

What Not To Do

```
class Position {
public:
    // ...

    // latitude
    double& getLatitude();

    // ...
private:
    double latitude;
    // ...
};

    Position pos;
    // ...
    pos.getLatitude() = 5.4;
```

- Non-const reference takes `const` off of the method
- It's pretty much the same as making the field public
- Very few applications where this makes sense
- Specially a problem in Java, as Objects are not passed by value but by Object Reference, and there isn't a `const`
- Instead: Always return data by value or const reference

As Far As A Mutator

```
/* Set position
  @param latitude Requires -90 <= latitude <= 90
  @param longitude Requires -180 < longitude <= 180
  @return if position is legitimate
*/
bool Position::set(double newLatitude, double newLongitude) {

    // valid latitude is -90 <= latitude <= 90
    if (newLatitude < -90 || newLatitude > 90)
        return false;

    // valid longitude is -180 < longitude <= 180
    if (newLongitude < -180 || newLongitude > 180)
        return false;

    latitude = newLatitude;
    longitude = newLongitude;

    return true;
}
```

Information Hiding Rule 2

Don't expose the difference between stored data and derived data

- Derived data is data calculated from stored data
- Don't reveal whether an attribute is stored or derived
- Use get method names for property methods
- get method names: `speed()`, `getSpeed()`
- property method names: ~~calculateS~~speed(), ~~determineS~~speed()

Position v2 Position::distance()

```
// distance from first to second position
double Position::distance(const Position& position) {

    // convert to Radians
    const auto long1 = d2r(longitude);
    const auto lat1 = d2r(latitude);
    const auto long2 = d2r(position.longitude);
    const auto lat2 = d2r(position.latitude);

    // Haversine Formula
    auto ans = pow(sin(((lat2 - lat1)) / 2), 2) +
               cos(lat1) * cos(lat2) *
               pow(sin((long2 - long1) / 2), 2);
    ans = 2 * asin(sqrt(ans)) * R;

    return ans;
}
```

- Changing to radians internally would make everything much simpler

- Do not want to change the current interface (i.e., in degrees), as clients expect it

- However, we could add radians to the interface while preserving what we have

Position v2 ⟶ v3

```
class Position {
    // ...
private:
    double latitude;
    double longitude;
};
```

```
class Position {
    // ...
private:
    double theta;
    double phi;
};
```

Position v2 & v3 Interface

```
class Position {
public:

    /* Set position

       @param latitude Requires -90 <= latitude <= 90
       @param longitude Requires -180 < longitude <= 180
       @return if position is legitimate
    */
    bool set(double latitude, double longitude);

    // latitude
    double getLatitude() const;

    // longitude
    double getLongitude() const;

    // distance to a position
    double distance(const Position& position);

    // heading to a position
    double heading(const Position& position);

private:
    double latitude;
    double longitude;
};
```

```
class Position {
public:

    /* Set position

       @param latitude Requires -90 <= latitude <= 90
       @param longitude Requires -180 < longitude <= 180
       @return if position is legitimate
    */
    bool set(double latitude, double longitude);

    // latitude
    double getLatitude() const;

    // longitude
    double getLongitude() const;

    // distance to a position
    double distance(const Position& position);

    // heading to a position
    double heading(const Position& position);

private:
    double theta;
    double phi;
};
```

Position v2 & v3 Client

```
// position of office
Position office;
office.set(36.538611, -121.797500);

// position of local coffee shop
Position coffeeShop;
coffeeShop.set(36.539722, -121.907222);

// calculate distance and heading from office to coffee shop
auto coffeeDistance = office.distance(coffeeShop);
auto coffeeHeading  = office.heading(coffeeShop);
```

Position::distance() v2 ⟶ v3

```cpp
// distance from first to second position
double Position::distance(const Position& position) {

    // convert to Radians
    const auto long1 = d2r(longitude);
    const auto lat1 = d2r(latitude);
    const auto long2 = d2r(position.longitude);
    const auto lat2 = d2r(position.latitude);

    // Haversine Formula
    auto ans = pow(sin(((lat2 - lat1)) / 2), 2) +
               cos(lat1) * cos(lat2) *
               pow(sin((long2 - long1) / 2), 2);
    ans = 2 * asin(sqrt(ans)) * R;

    return ans;
}
```

```cpp
// distance from first to second position
double Position::distance(const Position& position) {

    // Haversine Formula
    auto ans = pow(sin(((position.phi - phi)) / 2), 2) +
               cos(phi) * cos(position.phi) *
               pow(sin((position.theta - theta) / 2), 2);
    ans = 2 * asin(sqrt(ans)) * R;

    return ans;
}
```

Position:getLatitude() v2 ⟶ v3

```
// latitude
double Position::getLatitude() const {

    return latitude;
}
```

```
// @get latitude
double Position::getLatitude() const {

    return r2d(phi);
}
```

New Requirement

```
Route coffeeTrip(2);
coffeeTrip.setPosition(0, office);
coffeeTrip.setPosition(1, coffeeShop);
coffeeTrip.setPosition(2, office);
```

- Need to handle a multi-position route

Route v0

```
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // All positions on the route
    std::vector<Position>& getPositions();

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

Information Hiding Rules

1. Don't expose data items

2. Don't expose the difference between stored data and derived data

3. **Don't expose implementation details of a class**

4. **Don't expose a class's internal structure**

Route v0 Issues

```
Route coffeeTrip(2);
coffeeTrip.setPosition(0, office);
coffeeTrip.setPosition(1, coffeeShop);
coffeeTrip.setPosition(2, office);

// change first part of trip to coffee shop
coffeeTrip.getPositions()[1] = office;
```

- Can directly access elements of an internal container

- *Can we change the container?*

- *Can we change the container values?*

Route v0 & v1 Interface

```cpp
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // All positions on the route
    std::vector<Position>& getPositions();

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

```cpp
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // All positions on the route
    const std::vector<Position>& getPositions();

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

Information Hiding Rules

1. Don't expose data items

2. Don't expose the difference between stored data and derived data

3. **Don't expose implementation details of a class**

4. **Don't expose a class's internal structure**

Route v1 Issues

```
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // All positions on the route
    const std::vector<Position>& getPositions();

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

- Exposes direct design detail that we are using a `std::vector`
- If we change to a different container, the type returned by the `getPositions()` has to change, and the client code has to change
- Use of `auto` in the client code, and perhaps a `typedef` can help, but not entirely prevent this
- Which container we use is an *internal structure* and *implementation detail* that should be hidden in the class
- *Do we need this type of access?*

Route v1 & v2 Interface

```cpp
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // All positions on the route
    const std::vector<Position>& getPositions();

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

```cpp
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // Size of the route
    int size() const;

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

Route v2 Issues

```
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // Size of the route
    int size() const;

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

- The `setPosition()` exposes direct design detail that we are using an indexable container, e.g., `std::array`, `std::vector`, and `std::deque`, with an *indexable iterator*
- If we change to a container that does not allow indexing, the client code would have to change, e.g., `std::list`, `std::forward_list`, with a *bidirectional iterator*
- The characteristics of which container we use is an *internal structure* and *implementation detail* that should be hidden in the class
- Alwa
of ac

Route v2 & v3 Interface

```
class Route {
public:

    // constructor
    Route(int segments);

    // Set position at index
    void setPosition(int index, const Position& position);

    // Get position at index
    Position getPosition(int index);

    // Size of the route
    int size() const;

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

```
class Route {
public:

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Get position at index
    Position getPosition(int index);

    // Size of the route
    int size() const;

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

Route v3 Issues

```
class Route {
public:

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Get position at index
    Position getPosition(int index);

    // Size of the route
    int size() const;

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

- The `getPosition()` is expected to be a *O(1)* or *constant-time algorithm*
- However, with a non-indexable container, e.g., `std::list`, this is not true
- If we change a container that does not allow indexing, the client code would have to change
- The characteristics of which container we use is an *internal structure* and *implementation detail* that should be hidden in the class
- Always ask yourself *Do we need this type of access?*

Route v3 & v4 Interface

```
class Route {
public:

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Get position at index
    Position getPosition(int index);

    // Size of the route
    int size() const;

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

private:
    std::vector<Position> positions;
};
```

```
class Route {
public:

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

    // Restart iteration at the first position
    void firstPosition();

    // Advance to the next position
    void nextPosition();

    // Get current position
    const Position& currentPosition() const;

    // Are we done yet?
    bool isDonePosition() const;

private:
    std::list<Position> positions;
};
```

Route v4 Improvements

```
class Route {
public:

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

    // Restart iteration at the first position
    void firstPosition();

    // Advance to the next position
    void nextPosition();

    // Get current position
    const Position& currentPosition() const;

    // Are we done yet?
    bool isDonePosition() const;

private:
    std::list<Position> positions;
};
```

- Use standard C++ iterators

Route v4 & v5 Interface

```
class Route {
public:

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

    // Restart iteration at the first position
    void firstPosition();

    // Advance to the next position
    void nextPosition();

    // Get current position
    const Position& currentPosition() const;

    // Are we done yet?
    bool isDonePosition() const;

private:
    std::list<Position> positions;
};
```

```
class Route {
public:

    typedef std::list<Position>::const_iterator const_iterator;

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

    // Iterator to first position
    const_iterator cbegin();

    // Iterator past last position
    const_iterator cend();

private:
    std::list<Position> positions;
    decltype(positions)::const_iterator pposition;
};
```

Route v5 Additional Improvements

```
class Route {
public:

    typedef std::list<Position>::const_iterator const_iterator;

    // constructor
    Route();

    // Append position to route
    void append(const Position& position);

    // Distance of the specified segment number
    double distance(int segment);

    // Distance of the entire route
    double distance();

    // Heading for this segment
    double heading(int segment);

    // Iterator to first position
    const_iterator cbegin();

    // Iterator past last position
    const_iterator cend();

private:
    std::list<Position> positions;
    decltype(positions)::const_iterator pposition;
};
```

- A separate iterator object would allow multiple, simultaneous access
- A `begin()` and `end()` would allow use in range-for statement
- The `segment` is problematic. Perhaps allow iteration through *segments* and not positions