

Implementing Classes and Interfaces

Steven R. Bagley

Classes

- Objects have:
 - State
 - Operations
 - Identity
- Classes define State and Operations

Classes

- Two parts
 - Private implementation
 - Public interface
- Two users
 - Class Designer
 - Class User

Interface and Implementation

- Important to separate the two
- Rewrite implementation without rewriting the rest of the code
- Encapsulation

Interface

- Class user only sees the Interface (in theory)
- Interface should be understandable on its own

Interface Design

- Interfaces should be:
 - Cohesive
 - Primitive
 - Complete
 - Convenient
 - Consistent

Interface Cohesion

- A Class describes a single abstraction
- Logical group of operations
- Going back to the Mailbox example...

```
class Mailbox
{
public:
    void AddMessage(Message);
    char *GetCommand();
    void RemoveMessage(int);
    void GetMessage(int);
    void PrintMessage(int);
    void ListMessages();
    int Count();
}
```

```
class Mailbox
{
public:
    void AddMessage(Message);
    char *GetCommand();
    void RemoveMessage(int);
    void GetMessage(int);
    void PrintMessage(int);
    void ListMessages();
    int Count();
}
```

Interface Cohesion

- Most operations on the Mailbox class group together
- `GetCommand()` sticks out like a sore thumb
- Better to move it into a different class

Primitive Operations

- Operations shouldn't be decomposable to smaller operations
- `PrintMessage` is an example
 - Gets the message
 - Then prints it

Primitive Operations

```
bool List::advance(int& x)
```

- Purpose:
 - Advance cursor to next element
 - Reports if able to advance
 - sets `x` to the new cursor position
- Each of these should be a separate operation

Interface Completeness

- Define all operations that make sense on a class
- Even if they are not necessary for the current problem

Interface Convenience

- Don't be afraid to have non-primitive operations
- If they make things more convenient for the programmer
- E.g. File seeking operations
(could just close file and read in necessary bytes)

Interface Consistency

- Syntax basically
- Operation Names are the same:
 - `SetHeight()` and `setWidth()` not,
 - `set_height()` and `putWidth()`
- Similar parameters

Interface Consistency

- Bad example, C File I/O
- Based around FILE 'object'

```
fprintf(FILE, "Hello World");
```

```
fgets(str, 512, FILE)
```

Common Operations

- Several common types of operations
- Not all operations fit into these groups
- Not all classes will use all these groups

Constructors

- Called when object created
- Ensures that the object is initialized properly
- Otherwise random data
- Every class should have at least one constructor, preferably a default one

Destructor

- Called when object destroyed
- Ensures any resources are released properly
- Not present in Java

Accessors

- Returns a value from the object
- Does not modify the object at all
- Varying complexity

Accessor might cache calculations etc
Might take arguments

Mutators

- Any operation that might modify the object
- They are the *only* way an object gets changed
- Objects without mutators are *Immutable*
- Ensure object is still valid after change

Comparison Operators

- Compares an object with another of the same class
- Test whether equal, less than or greater than
- Specialist accessor

Iterators

- For classes that manage collections (such as a Linked List)
- Iterators provide mechanisms for moving about the collection
- Tend to be a mixture of *accessors* and *mutators*

Copying and Cloning

- Some objects can be copied very simply
- Others need more care
- Copy Constructors
- Clone methods

I/O

- Output operation renders the object into some form (e.g. to screen or as a string)
- Input operation sets fields of the object based on input (e.g. from keyboard or as a string)
- Input == Mutator, Output == Accessor

Design Hints

- Too many operations?
- One way, and only one way...
- Is it reusable in another context?
Don't over-engineer classes...
- On what else does the class depend
- No class is perfect

Class Unfit

- What to do when a class doesn't support the action you need?
- Or why is this data private — it'd be much easier if I could just...
- What do you do?

Class Unfit?

- Don't make the data public...
- Revisit the design
- Add the task as an operation
- Add accessors or mutators for the data

Would you be embarrassed by the changes to the interface in 6 months time...

Date Class

- Date Objects represent *valid* dates
- Access the date
- Change the date
- User can advanced them onto the next day
- Print

Date interface

- Can start to define the interface now
- Constructor
- Need *Accessors* for Day, Month and Year
- Need *Mutators* for Day, Month and Year

Date Mutators

- Date Objects should always be a valid date
- Independent Mutators could let the user set invalid dates

```
Date d(24,2,2008);
```

```
d.SetDay(30);
```

```
d.SetMonth(1);
```

```
d.SetYear(2009);
```

C++

```
Date d =
```

```
new Date(24,2,2008);
```

```
d.SetDay(30);
```

```
d.SetMonth(1);
```

```
d.SetYear(2009);
```

Java

```
Date d(24,2,2008);
```

```
d.SetDay(30); ←
```

```
d.SetMonth(1);
```

```
d.SetYear(2009);
```

C++

```
Date d =
```

```
new Date(24,2,2008);
```

```
d.SetDay(30); ←
```

```
d.SetMonth(1);
```

```
d.SetYear(2009);
```

Java

```
Date d(24,2,2008);
```

```
d.SetDay(30); ←
```

```
d.SetMonth(1);
```

```
d.SetYear(2009);
```

d contains 30/2/2009

C++

```
Date d =
```

```
new Date(24,2,2008);
```

```
d.SetDay(30); ←
```

```
d.SetMonth(1);
```

```
d.SetYear(2009);
```

Java

Date Mutators

- Date Objects should always be a valid date
- Individual Mutators could let the user set invalid dates
- Therefore, one *Mutator* for the whole date makes more sense

Date Interface

Constructor	Accessors	Mutators
Date(day, month, year)	Day	SetDate
Default?	Month	Advance
	Year	
	ToString (was Print)	

Date Implementation

- Day — number between 1 and 31
- Month — number between 1 and 12
- Year — open ended
- store all as an `int`

```
class CDate
{
public:
    CDate(int day, int month, int year);

    void Advance(int n);
    int Day();
    int Month();
    int Year();
    void SetDate(int day, int month,
                 int year);

    char* ToString();

private:
    int m_day;
    int m_month;
    int m_year;    C++
};
```

```
public class CDate
{
    public CDate(int day, int month, int year)
    { ... }

    public void Advance(int n) { ... }
    public int Day() { ... }
    public int Month() { ... }
    public int Year() { ... }
    public void SetDate(int day,
                        int month,
                        int year) { ...}

    public String ToString() { ... }

    private int m_day;
    private int m_month;
    private int m_year;
}
```

JAVA

```
class CDate
{
public:
    CDate(int day, int month, int year);

    void Advance(int n);
    int Day();
    int Month();
    int Year();
    void SetDate(int day, int month,
                int year);

    char* ToString();

private:
    int m_day;
    int m_month;
    int m_year;    C++
};
```

```
class CDate
{
public:
    CDate(int day, int month, int year);

    void Advance(int n);
    int Day() const;
    int Month() const;
    int Year() const;
    void SetDate(int day, int month,
                int year);

    char* ToString() const;

private:
    int m_day;
    int m_month;
    int m_year;    C++
};
```

Constant Operations

- Some operations (such as Accessors) do not alter the object
- `const` tells the compiler that this member function is read-only
- If Operation alters the state, the compilation will fail

Constructor

- Implementation is easy
- Check whether parameters is valid
- If valid, set member variables

Accessors

- Trivial...

```
int CDate::Day() { return m_day; }
```

```
int CDate::Month() { return m_month; }
```

```
int CDate::Year() { return m_year; }
```

Accessor

- `ToString()` similar
- Memory management issues in languages such as C++

Mutators

- `SetDate(...)` identical to the constructor
- `Advance()` slightly trickier
- Number of days in a month varies with month (and year for February)
- Code must take this into account
- Or use an alternative method

Advance Mutator

- Julian Date format
- Number of days since Jan 1, 4713 BC
- Convert date to this format
- Add days
- Convert back

Static Member Functions

- Need routines to convert between DMY and Julian representations
- Not really related to instances
- But related to class
- Use Static Member functions

Adding functionality

- Class users find they need to:
 - Compare dates
 - Find how many days between dates
- Probably don't need to redesign class
- Functionality that belongs to the class
- Add the operations to the class interface

Implementation

- New Operations require conversion to and from Julian format
- Would we be better changing the class implementation to use Julian dates?
- No need to change interface so class users unaware (except perhaps for speed boost!)

```
int CDate::Day() { return m_day }
```

Day, Month, Year

```
int CDate::Day()  
{  
    int day, month, year;  
    Julian2DMY(m_julDate, &day, &month,  
    &year);  
  
    return day;  
}
```

Julian

Classes, not basic types

- Objects should be the norm
- Basic types should be rare
- Group related basic types into sub-objects

```
class CStyledText
{
public:
    ...
private:
    char *m_textToDisplay;
    float m_xLocation;
    float m_yLocation;
    char *m_fontName;
    float m_pointSize;
    bool m_bold;
    bool m_italic;
};
```

C++

```
class CFontStyle
{
public:
    ...
private:
    char *m_fontName;
    float m_pointSize;
    bool m_bold,
        m_italic;
};
```

```
class CPoint
{
public:
    ...
private:
    float m_xLocation;
    float m_yLocation;
};

class CStyledText
{
public:
    ...
private:
    char *m_textToDisplay;
    CPoint m_location;
    CFontStyle m_font;
};
```

Class hints

- Fields do not always need accessors/
mutators
- Generally not needed if the field is part of
the implementation

Class hints

- Declare `Accessors` `const`
(that includes you Dr. Bagley...)
- Use a Canonical form for your classes

Canonical Form

- Give the users easy access to the information they want
- Public first, then private
- Users are interested in the public interface, not the implementation

Canonical Form

- Within the public section
 - List the constructors
 - List the mutators
 - List the accessors
- Within the private section
 - List private operations then the fields

How do I make an object?

What can I do with it?

How can I find out the result?

Contracts

- Need to know more than just how to call an interface
- Need to know what assumptions are involved in calling the method
- A *Contract* between the client and the object

Contract Example

- Take a class representing a stack of integers
- What happens if we try and pop an object off an empty stack?
- Many options, programmer needs to know what the stack type does

Contracts

- **Preconditions** — must be true before a method is called or the result is undefined
- **Postconditions** — are true after a method is called
- **Class Invariants** — things that are true after the completion of the constructor and before and after every method call

Preconditions

- Back to the stack example
- Suppose we apply the precondition to pop that the stack must not be empty
- This means that the client code is responsible for checking that the stack isn't empty before trying to pop values
- Must be verifiable

Defensive Programming

- Alternative to preconditions is that the code checks the values on each method call
- More secure — object never gets into an invalid state
- Less efficient — checks may be unnecessary

Post-Conditions

- Post-condition for a stack's push method is that the stack is not empty
- If post-conditions match to pre-conditions, no need to check the state (we know it to be the case)

```
SomeStack *stk;
```

```
stk->push(42); // POST: Stack is now non-empty  
stk->pop(); // Last line leaves stack in a  
             non-empty state which matches  
             the pre-condition for pop
```

Contracts and Interfaces

- Any object that implements an interface must follow the same contract as the interface defines
- Care needs to be taken that you don't break the contract