Lecture notes – "Writing hi-quality code"

Gustav Taxén, Propellerhead Software
gustav.taxen@propellerheads.se
www.propellerheads.se

Single responsibility principle

Let each class be responsible for one thing only.

Example:

```
class CRectangle {
  public: CRectangle(
    float iLeft,
    float iTop,
    float iRight,
    float iBottom);

  public: void Draw(CWindow&) const;
  public: float Area() const;
};
```

This class has two responsibilities; it can draw itself and it can compute the area. If we want to use the class (as a mathematical representation of a rectangle) in some other context, we have to include everything that has to do with drawing (e.g., CWindow) too.

Having multiple responsibilities makes classes harder to refactor. When code gets difficult to work with it's usually because classes have multiple responsibilities.

Fragile base class problem

When a class inherits from a concrete (non-abstract) base class and do not override all methods in the parent class you may end up with a fragile base class problem. This happens when the subclass makes assumptions about the base class that aren't correct (or if someone changes the base class so that the subclasses' assumptions are no longer correct).

Example:

```
class <typename T> CStack : public std::vector<T> {
  public: CStack() : fIndex(0) {}
  public: void Push(const T& iElement) {
    push_back(T);
    ++fIndex;
  }
```

```
  public: T Pop() {
    std::vector<T>::iterator i = begin();
    i = i + fIndex;
    erase(i);
    --fIndex;
  }
  private: std:size_t fIndex;
};
```

This stack class inherits the standard `vector` class but forgets to override its `clear()` method. If someone calls this method on an instance of the stack class, the stack will be incorrect.

Avoid inheriting from non-abstract base classes! A drastic approach is to *only* implement interfaces and never use inheritance at all.

Design by contract

Designing by contract is to have explicit, verifiable inputs and outputs to all functions in your code.

Example:

```
/* Compute square root. Input must not be negative. */
float sqrt(float a) {
  assert(a >= 0);
  float result = …
  assert(!isinf(result));
  assert(!isnan(result));
}
```

In the example, the contract for the function is that the input should be non-negative and that the result should be a non-infinite real number.

Verifying contracts using `assert` is a very cheap way to catch difficult bugs. Make a habit to always use design by contract – you won't regret it! (This has no impact on the performance of the final code. When you compile for release, all assert statements will be removed automatically by the compiler.)

Invariant checking

The invariant of a class instance is one or several statements that should always be true, regardless of which changes you make to the instance. All classes have at least one invariant that can be checked (that the instance exists) but there are often other invariants too.

Example:

```
class CRectangle {
```

```
  public: CRectangle(
    float iLeft,
    float iTop,
    float iRight,
    float iBottom)
  :
    fLeft(iLeft),
    fTop(iTop),
    fRight(iRight),
    fBottom(iBottom)
  {
     assert(CheckInvariant());
  }

  bool CheckInvariant() const {
    assert(this != NULL);
    assert(fLeft <= fRight);
    assert(fTop <= fBottom);
    return true;
  }
};
```

Verifying the invariant of a class whenever a class method is called (and just before a non-const method returns) is a very cheap way to catch difficult bugs. Make a habit of checking the invariant – you won't regret it!

Object lifetime management

It is very important to think about the lifetime of your class instances. It doesn't matter if your language has a garbage collector!

Example:

```
class CGame;
class CRenderer {
  public: CRenderer(CGame* iGame);
  private: CGame* fGame;
};
class CGame {
  public: CGame();
  public: void SetRenderer(CRenderer* iRenderer);
  private: CRenderer* fRenderer;
};
...
CGame* game = new CGame;
CRenderer* renderer = new CRenderer(game);
game->SetRenderer(renderer);
...
```

In this example, both classes need to know about each other. This will probably lead to difficulties when the instances should be destroyed or replaced. You would have to think carefully about the order of creation and the order of destruction.

Strong coupling between classes is often a result of a non-optimal design. Perhaps the classes have too many responsibilities? If not, then can you create another class (e.g., a factory class) that allows you to encapsulate the object lifetime problems?

Exception safety

An exception is a run-time error that cannot be caught at compile-time. Examples include out-of-memory errors, file-not-found errors, etc.

When you call a function, try to be aware of the level of exception safety it has:

- **No-throw guarantee** – The function or method will never throw an exception.
- **Strong exception safety** – The function or method may fail but the failure has no side effects. Everything is exactly the way it was before the exception was thrown.
- **Basic exception safety** – The function or method may fail and there may be side effects, but all invariants are preserved and no resources are leaked.
- **No exception guarantees** – The function or method may fail, and anything may happen.

Note that very few functions have a no-throw guarantee. For example, the std::string class in C++ allocates memory on the heap, which means that many of the operations on this class may throw an exception. It is safest to always assume that all function calls may throw an exception.

Always aim for strong exception safety in your code! If, for some reason, you cannot guarantee strong exception safety, you must at least provide basic exception safety.

Immutable objects

An immutable object is a class instance that cannot be modified after it has been created.

Example:

```
class CRectangle {
  public: CRectangle(
    float iLeft,
    float iTop,
    float iRight,
```

```
    float iBottom);

  public: float GetLeft() const;
  public: float GetTop() const;
  public: float GetRight() const;
  public: float GetBottom() const;
};

float Area(const CRectangle& r);

CRectangle Intersect(
  const CRectangle& a,
  const CRectangle& b);
```

This class has no methods that can change it – all you can do is to create an instance with a given set of coordinates, and access those coordinates after creation. All functions that operate on rectangles are free functions: the intersection operation, for example, takes two rectangles as input and returns a new rectangle that is the result of the operation.

Immutable classes have many advantages compared to mutable classes: they are simple, their methods never have any side effects, it is easy to implement the strong exception guarantee (or no-throw) for them, object lifetime management becomes easier, their contracts and invariants are usually simple, the code that operates on them becomes easier to profile and analyze, and they are automatically thread-safe!

Don't create methods that mutate your classes unless you absolutely have to!

Code simplicity

The following example is cool, but almost impossible to understand. It is (alledgedly) from the game *Quake 3*. It computes 1/sqrt(x) using Newton-Raphson. It relies on the way floating point numbers are stored in (most) computers (http://en.wikipedia.org/wiki/Fast_inverse_square_root).

```
float InvSqrt (float x){
    float xhalf = 0.5f*x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i>>1);
    x = *(float*)&i;
    x = x*(1.5f - xhalf*x*x);
    return x;
}
```

Always aim for writing readable, simple code! Never optimize away legibility unless you absolutely have to (and always document the new, complex code).

General tips

Here are some general tips for improving code quality:

- Always ensure that your code has zero compiler warnings. If there's a warning, it's usually an indication that something's wrong or dangerous.
- Agree upon a set of formatting guidelines with your team and stick to them consistently! It will make your code easier to maintain and read.
- Always write a brief comment in the code that states what a class or function do. If you design by contract (and you should!), document the contract too.
- Ask a teammate to read and review your code check-ins! He/she will find lots of errors and opportunities for improvements that you never saw yourself!

Testing

Tests can be broadly classified into these categories:

- **Unit test** – tests a specific contract
- **Integration test** – tests interoperation between different classes or modules
- **Regression test** – tests that a previous bug that was fixed hasn't re-appeared
- **Acceptance test** – tests that a feature is correctly implemented according to specification
- **Smoke test** – quick test, just to assert that nothing "blows up" when you start the program or use a feature

Unit tests can be automated. There are many unit testing frameworks available, for all programming languages. Use one!

It is important to have unit tests for the basic, most fundamental classes in your system (since you're building upon them). Writing unit tests for higher-level modules can be difficult; you may have to resort to integration testing for those (for example, the GUI of a program is often tricky to unit-test since it normally has very complex invariants).

Consider using test-driven design to write your most critical classes!

Recommended reading

"Code Complete", Steve McConnell
"Effective C++", Scott Meyers
"Exceptional C++", Herb Sutter
"Writing Solid Code", Steve Maquire