Advanced Software Engineering with C++ Templates

Lecture 7: LSP, Exceptions

Thomas Gschwind <thg@zurich.ibm.com>
Bitte bis 4. Dezember ausfüllen...

http://t.uzh.ch/9R7HT
Agenda

- The Liskov Substitution Principle
  - Square → Rectangle
  - Rectangle → Square

- Exceptions
Squares and Rectangles

- I had a colleague who was unsure whether to derive rectangle from square or vice-versa. What would you recommend to him? Implement a set of sample programs illustrating various options and your recommendation! The programs should demonstrate why your solution is better than the other solutions.
Squares and Rectangles

- Class hierarchy for a vector graphics program
- We have an abstract class Shape from which various geometric shapes are being derived
- Task:
  - Add the following classes: Square, Rectangle
  - Also add useful members like {get,set}_{width,height,length,size}, ...
Liskov Substitution Principle

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

Put Simple: Every function/program must work the same when it is invoked with a subclass instead of the expected class. This is the responsibility of the subclasses!
class Rectangle {
    int w, h;

public:
    virtual void setWidth(int wi) { w=wi; }
    virtual void setHeight(int he) { h=he; }
};

class Square : public Rectangle {
public:
    virtual void setWidth(int w) {
        Rectangle::setHeight(w);
        Rectangle::setWidth(w);
    }

    virtual void setHeight(int h) {
        setWidth(h);
    }
};
Square $\rightarrow$ Rectangle (cont’d)

- What happens if we pass a Square to a function test expecting a Rectangle?
- Can we expect the function test to know about Square?
- Would it be OK if the function yields an unexpected outcome under the motto garbage-in garbage-out?
- No! Because we claim that a Square is a Rectangle!

```cpp
void test(Rectangle &r) {
    r.setWidth(5);
    r.setHeight(4);
    assert((r.getWidth() * r.getHeight()) == 20);
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
class Rectangle {
    int w, h;

public:
    void setWidth(int wi) { w=wi; }
    void setHeight(int he) { h=he; }
};

class Square : public Rectangle {
public:
    void setWidth(int w) {
        Rectangle::setHeight(w);
        Rectangle::setWidth(w);
    }
    void setHeight(int h) {
       setWidth(h);
    }
};

Th. Gschwind. Fortgeschrittene Programmierung in C++.
What happens if we pass a Square to a function test expecting a Rectangle?

The program works now as expected.

However, since set_width/set_height are no longer virtual our Square is now a Rectangle

=> This is not the solution either

```c++
void test(Rectangle &r) {
    r.setWidth(5);
    r.setHeight(4);
    assert((r.getWidth())*r.getHeight())==20);
}
```
Rectangle → Square

- Square has one field (height or width)
- Rectangle adds another one for the other dimension
  - Makes sense from a memory usage point of view
- Square has a method to set the length
- Rectangle adds methods for height and width
  - Also makes sense from the methods they provide
  - A Square does not need set_height & set_width
class Square {
    int l;
public:
    virtual void setLength(int le) { l=le; }
    virtual int getLength() { return l; }
};

class Rectangle : public Square {
    // reuse length as height
    int w;
public:
    virtual void setLength(int le) {
        Square::setLength(le); setWidth(le); }
    virtual void setHeight(int he) {
        Square::setLength(he); }
    virtual void setWidth(int wi) { w=wi; }
};
Rectangle $\rightarrow$ Square (cont’d)

- We cannot pass a Square to the test function

```cpp
void test(Rectangle &r) {
    r.setWidth(5);
    r.setHeight(4);
    assert((r.getWidth() * r.getHeight()) == 20);
}
```

- What if test would take Squares?
  - No problem either, LSP fulfilled

```cpp
void test(Square &s) {
    s.setLength(5);
    s.setLength(4);
    assert((s.getLength() * s.getLength()) == 16);
}
```
Let us extend our Shape class hierarchy

Our customer requests a `get_area()` member

```cpp
int Square::get_area() { return l*l; }
int Rectangle::get_area() { return l*wi; }

void foo(Square &s) {
    assert((s.get_length() * s.get_length())
           == s.get_area());
}
```
Rectangle → Square (cont’d)

- Let us extend our Shape class hierarchy
- Our customer requests a get_area() member

```cpp
int Square::get_area() { return l*l; }
int Rectangle::get_area() { return l*wi; }

void test(Square &s) {
    assert((s.get_length()*s.get_length())
           ==s.get_area());
}

void oh_no_not_another_test() {
    Rectangle r(3,4);
    test(r);
}
```
Square → Shape, Rectangle → Shape

Correct, don’t use deep class hierarchies just because it is “object-oriented” programming (is it?) or because it’s “cool” or “professional” (or whatever)!
Lessons Learned

- Avoid code inheritance
  - People will nail you down on how the base class is implemented
- If inheritance is needed define an interface
  - Now, only the documentation counts
- Inherit the interface not the code
- If code inheritance is needed, prefer the use of an interface plus use aggregation
  - Follow this advice especially when using Java
  - Yes, you need to type a bit more
- If this is really (really?) not an option provide both an interface and a class to inherit from
Agenda

- The Liskov Substitution Principle
  - Square $\rightarrow$ Rectangle
  - Rectangle $\rightarrow$ Square

- Exceptions
Some Java Trivia – finally

```java
public int foo(int x) throws Exception {
    if (x==0) return 0;
    if (x!=1) {
        try {
            if (x<4) throw new Exception("nix gut");
            if (x==4) return 4;
            return 5;
        } catch (Exception e) {
            if (x==-1) throw new Exception("gar nix gut");
            if (x==2) return 2;
        }
        finally {
            return 3;
        }
    }
    ........
    return 1;
}
```
Error Handling

Traditional techniques

- Terminate the program
  ```c
  exit(error_code);
  ```

- Special return value
  ```c
  return -1; or return NULL;
  ```

- Set a special variable (e.g., `errno` in C)
  - Can be useful if there is no “special return value”
  - Needs to be checked by caller after every call

- Callback routine
Exceptions

- Stack unwinding technique
  - Generated (thrown) as soon as an error is detected
  - Caught from an error handling “routine” (better block)
  - Integrated with destructor handling simplifying resource management

- Can be used for optimization (avoid this one)
  - Depends on architecture

- Some really completely and totally outdated 20 year old C++ compilers have problems with exceptions

- Be careful when mixing code compiled with a C and C++ compiler
Exceptions (cont’d)

```cpp
class MathErr {};
class ZeroDivide: public MathErr {};

fraction operator/(const fraction &a, fraction b) {  
    if (b.c==0) throw ZeroDivide();  
    swap(b.c,b.d);  return a*b;
}
```

```cpp
test.cc
fraction a(0), b, c;
// ...
try {
    r=b+a*sqrt(sq(b)-4*c))/(2*a);
} catch (ZeroDivide &zd) {
    // ...
}
```

Invokes destructors of variables leaving scope
Invokes directly the exception handler
More readable and more efficient
Standard Exceptions (std::exception)

- `std::bad_alloc` – thrown by `new` on allocation failure
- `std::bad_cast` – thrown by `dynamic_cast` when it fails with a referenced type
- `std::bad_exception` – thrown when an exception type doesn't match
- `std::bad_typeid` – thrown by `typeid`
- `std::ios_base::failure` – thrown by functions in the iostream library
Standard Exceptions (std::logic_error)

- Represents “predictable” errors and errors that should have been avoided in the first place
  - `domain_error` is not thrown by the standard library but may be used by third party libraries (e.g., boost::math uses it)
  - `invalid_argument` is thrown if an argument is not accepted (e.g., bitset, stoX)
  - `length_error` is thrown when we exceed implementation defined length limits (e.g., string, vector::reserve)
  - `out_of_range` is thrown by bound-checked members (e.g., at)
Standard Exceptions (std::runtime_error)

- Represents “unpredictable” errors
  - `overflow_error`, `underflow_error` are thrown on a math overflow (e.g., `bitset`, `boost::math`)
  - `range_error` is thrown when a value has an incorrect range (e.g., `wstring_convert::from_bytes`)

Th. Gschwind. Fortgeschrittene Programmierung in C++.
Optimizing with Exceptions

- Although as a construct to use with care, this sample is shown in Stroustrup’s the C++ Programming Language (3rd ed)
- I have seen it in a program
- My take, don’t do this; typically, checking for an empty queue or upfront the number of elements is not expensive

```cpp
void for_each(Queue &q) {
    try {
        for(;;) {
            int i=q.get(); // throws 'Empty' if empty
            // ...
        }
    } catch (Queue::Empty) {
        return;
    }
}
```
Exception Specification

- If no exceptions are specified, a routine can throw any exception

- Exceptions can be specified as part of the signature
  - `void func(...) throw (e1, e2, ...) { ... }
  - C++11: this feature has been deprecated due to “lack of success”: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3051.html

- If no exception is thrown, this needs to be specified explicitly
  - `void func(...) throw ()`
  - C++11: `void func(...) noexcept`
  - C++11, noexcept can take a `constexpr` as argument indicating whether the function may throw an exception (true) or not (false)
class Matherr {};
class ZeroDivide: public MathErr {};

fraction operator/(const fraction &a, fraction b) throw (ZeroDivide) {
  if (b.c==0) throw ZeroDivide();
  swap(b.c,b.d); return a*b;
}

test.cc

fraction a, b, c, r;
// ...
try {
  r=b+a*sqrt(sq(b)-4*c))/(2*a);
} catch (ZeroDivide &zd) {
  // ...
}
Exception Specification (cont’d)

Why is it optional in C++? Why is it deprecated in C++11?

▪ In general, the compiler cannot always check them
  • Multiple object files (from different sources)
  • Derived classes

▪ Handling unexpected exceptions
  • unexpected()
    (can be set with the set_unexpected() function)
  • Typically, this handler invokes std::terminate()
Exception Specification: Multiple Object Files

- Initially, all is fine
- What if, due to new requirements, your application needs V1.1 of Library Y
- Version 1.1 changes some of the exceptions
  - Yes, should not have happened (it did anyways)
- Options
  - Not implement the new requirement
  - Rewrite Library X (good luck if closed source)
  - Simply don’t take exceptions that serious (yes, this is ugly!)
Exception Specification: Mult. Object Files (cont’d)

- **Java**
  - Since Library X won’t be recompiled, all is “fine” (in the current Java version)
  - What if Library Y throws a new exception? Since we rely on the “old” exception specifications of Library X, it can happen that sometimes an exception not listed in Library X’s interface description is thrown
  - Probably our application does not catch the exception and the application is terminated.

- **C++**
  - That’s what `unexpected` is good for
  - However, handling the problem in `unexpected` has proven difficult/impossible which is one of the reasons this is deprecated in C++11
Exception Specification: Derived Classes

- The specification makes perfect sense
  - When writing to a file, an IO-Exception can occur

```java
public class Writer {
    // ...
    public void write(char[] buf) throws IOException {
        // ...
    }
    // ...
}
```
Sometimes, we don’t want to write to a file

What if the data generated needs to be processed again

- We may want to write the data to the memory and process it from there
- To achieve this, we derive a StringWriter from Writer

```java
public class StringWriter extends Writer {
    // ...
    public void write(char[] buf) throws IOException {
        // sometimes we need to allocate more memory
        // what if we are out of memory?
    }
}
```

We would love to, but no, we cannot throw an `OutOfMemoryException`
Exception Discrimination

- Inheritance – order of catch clauses
- Precisely the same as in Java
Exceptions (cont’d)

▪ Catch all?
  • C++ has no common base class that can be used for all classes or exceptions
  • Yes, one can define such a class oneself
    (be careful it may make your code worse than you may think)
  • `catch(...)` // yes three dots

▪ What can be used as exception?
  • Any data type

▪ Rethrowing an exception?
  • `throw;`

▪ Call/Catch by value vs. call by reference
  • Give preference to `T&` over `T`
public int foo(int x) throws Exception {
    if (x==0) return 0;
    if (x!=1) {
        try {
            if (x<4) throw new Exception("nix gut");
            if (x==4) return 4;
            return 5;
        } catch (Exception e) {
            if (x==-1) throw new Exception("gar nix gut");
            if (x==2) return 2;
        }
        finally {
            return 3;
        }
    }
    // return 6;
}
return 1;
finally

- Yes, finally poses interesting questions
- C++ does not have it
- Really?
- Really
- Really
- But C++ has something better! 😊
- You still think finally is a good idea
  - C++11 allows you to simulate it
Cleaning up in C++

- When is \texttt{s} allocated and when deallocated?
  - When it enters, respectively leaves the try block

- What is executed at these points?
  - The constructor and destructor

```cpp
void foo() {
  try {
    String s="Hello World";
    // problematic code
  } catch ( /*...*/ ) {
    // exception handler
  }
}
```
Cleaning up in C++ – The Solution

```cpp
struct DBLocker {
    string res;
    DBLocker(string lockme) : res(lockme) {
        lock(lockme);
    }
    ~DBLocker() { unlock(res); }
}

void foo() {
    try {
        DBLocker lock("whatever we want to lock");
        // problematic code
    } catch ( /*...*/ ) {
        // exception handler
    }
}
```

When this block is left, the destructor will be executed and our external resource will be unlocked.

Th. Gschwind. Fortgeschrittene Programmierung in C++.
C++11: Simulating finally

```cpp
template<typename F> struct final_action {
    F clean;
    final_action(F f) : clean(f) {};
    ~final_action() { clean(); }
}

template<class F> final_action<F> finally(F f) {
    return final_action<F>(f);
}

void foo() {
    try {
        auto finally_action = finally([&]{ /* handler */ });
        // problematic code
    } catch (/*...*/) {
        // exception handler
    }
}
```

Th. Gschwind. Fortgeschrittene Programmierung in C++.
C++11: Simulating finally

- Lambda function allows to specify the cleanup code within the function
  - In certain cases, this may be more readable
  - For instance, if no resource allocation/deallocation pair can be identified
- Implementation-wise, same approach as before
Summary

- The Liskov Substitution Principle
  - Square $\rightarrow$ Rectangle
  - Rectangle $\rightarrow$ Square

- Exceptions
Exercise 1: Range Checking Iterator

- Implement an iterator that encapsulates another iterator (i.e., a sequence) and that performs range checking.
- The iterator is initialized with the current element, and the first and last element of the sequence.
- If the iterator points to the first element and is decreased OR if the iterator points to the last element and is increased signal an error – choose an appropriate form of signaling the error.
Exercise 2: Stack and Inheritance

- Implement the Stack class with virtual members from the last lecture
  - Implement it with a limited stack size of 256 elements
- Create a subclass called UnlimitedStack
  - The unlimited stack should be able to store an unlimited number of elements
- How do you implement the class? How do you deal with the issues discussed in the last lecture?
Exercise 3: std::back_inserter

- Look at the std::back_inserter. Use this function to allow users to merge two containers.
Exercise 4: Locker Class

- Implement a small class such as the DBLocker class.
- Instead, it is called FileLocker to lock a file (see man 2 flock).
- Use that class in your RPN calculator
Next Lecture

- Factories in C++
- Standard Library
  - Input & Output
  - Regular Expressions

Have a nice weekend, see you in two weeks