Large Scale Software Design

Mario Heene

Universität Erlangen-Nürnberg

July 25, 2011

Motivation

Literature

- J. Lakos: Large Scale C++ Software Design, Addison-Wesley, 2007
- Summary in R. Martin: More C++ Gems, Addison-Wesley, 2000

"Experience gained from small projects does not always extend to larger development efforts."
– John Lakos

You need other materials and techniques to build a scyscraper than you need to build a single family home.

Motivation

Root causes for poor quality in large software systems

- Compile-time dependencies
- Link-time dependencies

Content

Excessive Compile Time Dependencies - Example

```
//myerror.h
#ifndef MY_ERROR_H
#define MY_ERROR_H

struct MyError{
    enum Codes{
        SUCCESS = 0,
        WARNING,
        IO_ERROR,
        //...
        READ_ERROR,
        WRITE_ERROR,
        //...
        BAD_STRING,
        BAD_FILENAME,
        //...
        MARTIANS_HAVE_LANDED,
        //...
    }
}

#endif
```

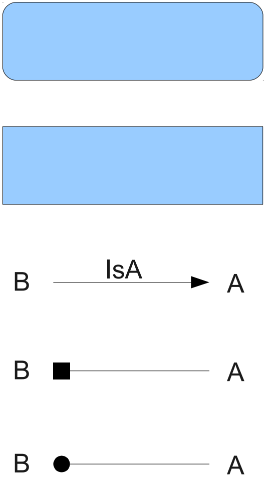
Listing 1: myerror.h

Excessive Compile Time Dependencies

- changing a header file might cause several translation units to recompile
- no problem in small projects (maybe you don't even notice)
- compile times may increase significantly when the project gets larger (days, weeks)

When you reach a point in time, where you don't make reasonable modifications to your header files any more, because of compilation time, then your whole design is useless and unmaintainable.

Notations



X is a logical entity (e.g. class)

X is a physical entity (e.g. file)

B is a kind of A

B uses A in B's interface

B uses A in B's implementation

Component

Logical View

- subset of the logical design that makes sense to exist as an independent cohesive unit
  $\rightarrow$ classes, functions, ...

Physical View

- indivisible physical unit: none of its parts can be used independently of the others
- consists of exactly one header (.h) file and one implementation (.c) file

- components are atomic units
- basic building blocks of our physical design

Component - Example

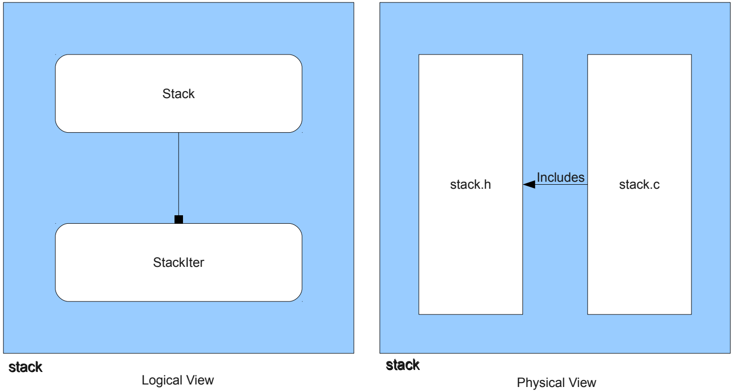


Figure: Logical and physical view of the stack component

The “DependsOn” Relation

A component **y** DependsOn a component **x** if **x** is needed to compile or link **y**.

- DependsOn is a physical relation between components (physical entities)
- different from logical relations like IsA or Uses which apply to logical entities

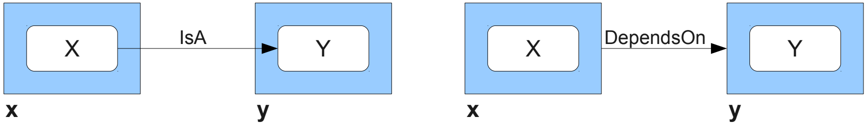


Figure: Logical vs. Physical Dependencies

Implied Dependencies

- logical designs imply physical dependencies
- can be predicted and analysed *before* any code is written

Compile-Time Dependencies

A component **y** exhibits a compile-time dependency on component **x** if **x.h** is needed to compile **y.c**

Link-Time Dependencies

A component **y** exhibits a link-time dependency on component **x** if **y.o** contains undefined symbols for which **x.o** must be called directly or indirectly at link-time.

Implied Dependencies - Example

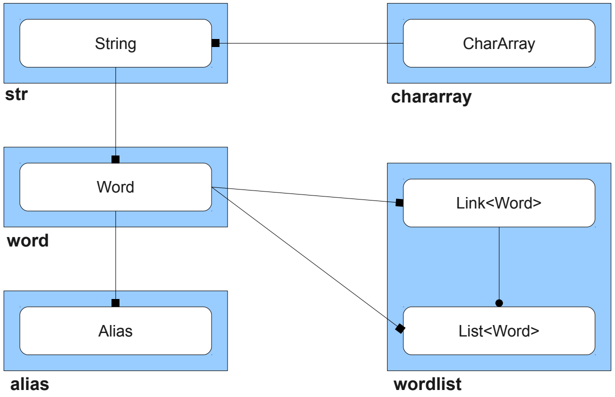


Figure: Logical Dependencies

Implied Dependencies - Example

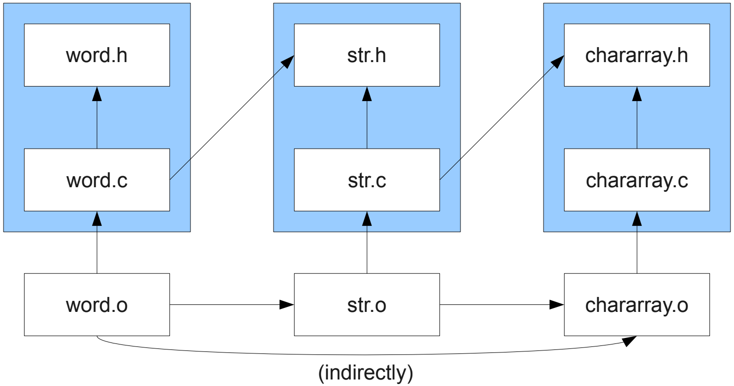
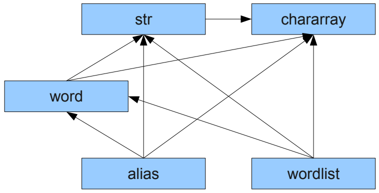
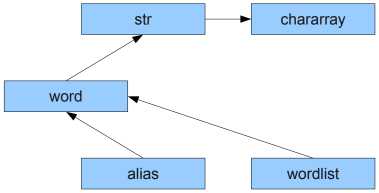


Figure: Compile-Time and Link-Time Dependencies

Implied Dependencies - Example



complete graph



graph without redundant edges

Figure: Physical Dependencies between Components

DependsOn relation for components is transitive

If component **x** DependsOn component **y** and **y** DependsOn component **z**, then **x** DependsOn **z**.

Content

Acyclic versus Cyclic Dependencies

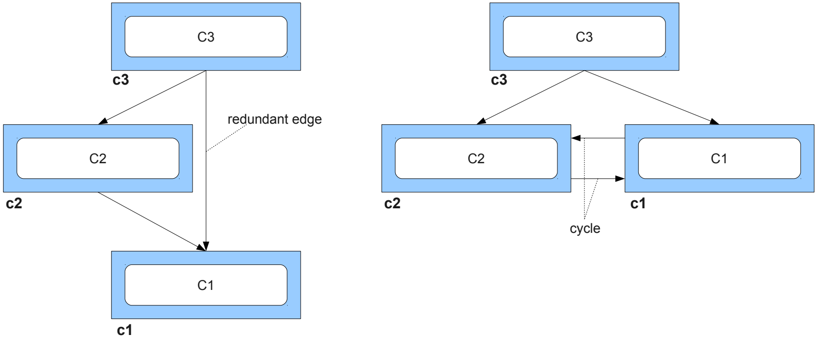
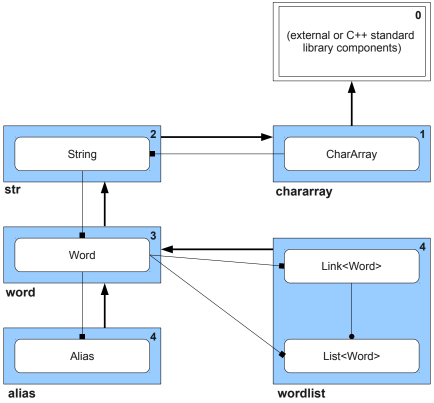


Figure: Acyclic versus cyclic dependencies

Which of the both designs do you think is better with respect to testing?

Level Numbers



A physical dependency graph that can be assigned unique level numbers is said to be levelisable.

$\rightarrow$ not possible with graphs that contain cyclic dependencies

Hierarchical and Incremental Testing

Hierarchical Testing

Assuming the components of the previous level are working properly, we can test the components at each level individually.

Incremental Testing

Testing only the additional functionality implemented within the component currently under test.

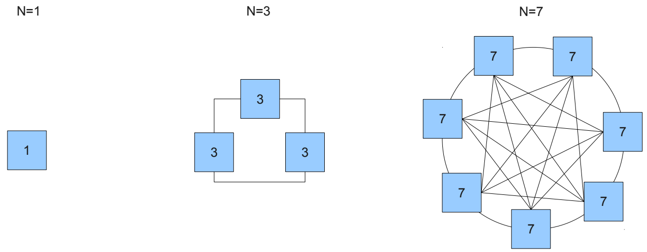
Levelisable designs should be preferred!

Cumulative Component Dependency

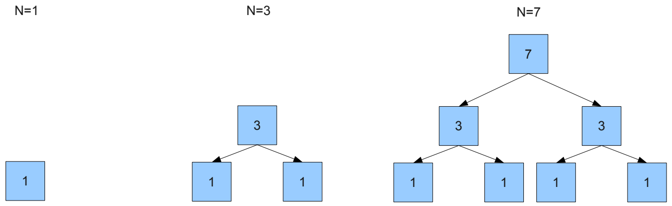
- CCD is a metric closely tied to link-time cost
- indicates relative costs associated with developing, testing, maintaining, and reusing a given subsystem

CCD is the sum over all components C_i of the number of components needed to test each C_i incrementally.

Cumulative Component Dependency



CCD	1	3*3=9	7*7=49



CCD	1	2*1+3=5	4*1+2*3+7=17

Cumulative Component Dependency

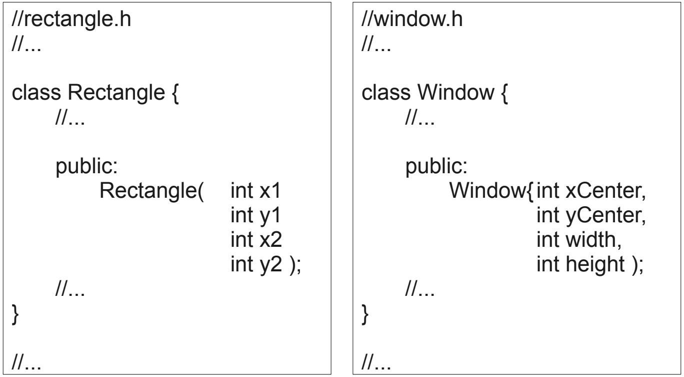
CCD scales with:

- $O(N^2)$ for purely cyclic designs
- $O(N \log(N))$ for perfect tree-like designs

Minimizing the CCD should be one of your design goals.

Content

Levelization



```
//rectangle.h
//...

class Rectangle {
    //...

    public:
        Rectangle(   int x1
                     int y1
                     int x2
                     int y2 );
    //...
}

//...
```

```
//window.h
//...

class Window {
    //...

    public:
        Window{int xCenter,
               int yCenter,
               int width,
               int height );
    //...
}

//...
```

Figure: Two representations of a box

Levelization

```
//rectangle.h
//...

class Rectangle {
      //...
      public:
            Rectangle(   int x1
                         int y1
                         int x2
                         int y2 );

            Rectangle( const Window& w );
      //...
}

//...
```

```
//window.h
//...

class Window {
      //...
      public:
            Window{int xCenter,
                   int yCenter,
                   int width,
                   int height );

            Window( const Rectangle& r );
      //...
}

//...
```

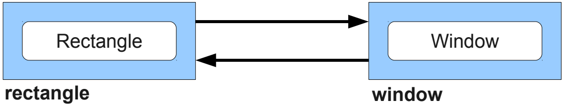


Figure: Mutually dependant components

Levelization

```
//boxutil.h
//...

class BoxUtil {
      //...
      public:
            static Rectangle   toRectangle( const Window& w );

            static Window       toWindow( const Rectangle& r );
      //...
}

//...
```

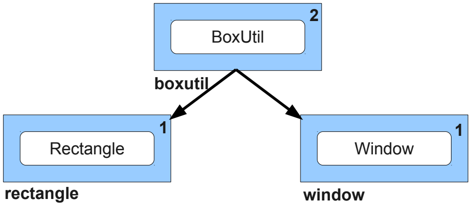


Figure: Cyclic dependencies resolved by escalation

Insulation

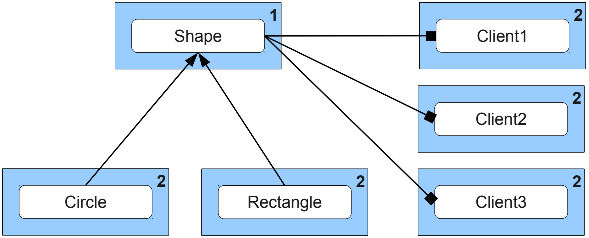
Header file usually contains:

- public interface by which functionality of the class can be accessed
- private members that are needed for the implementation

Problem: Changing the implementation might cause dependant classes to recompile when private members of the header file are changed.

Insulation

```
//shape.h
//...

class Shape {
      public:
            Shape( int x, int y );

      private:
            int _x;
            int _y;
}

//...
```



Changing for instance the type of **_x** and **_y** to **short int** would force all other components to recompile (including the client components).

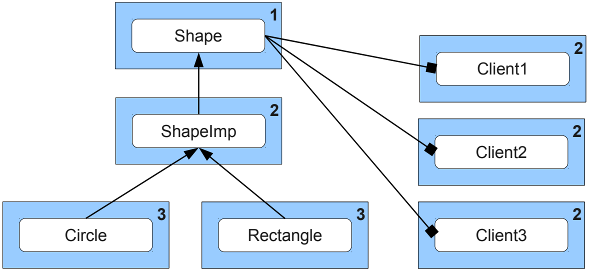
Insulation

An implementation detail (like a type, data, function) is said to be insulated if it can be changed, removed or added without forcing clients of the component to recompile.
Insulation is the technique of producing an insulated implementation.

Insulation is the physical analogue to encapsulation.

Insulation – Protocol class

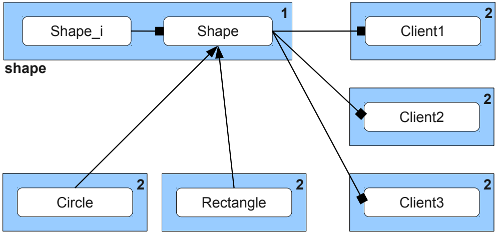
```
//shapeimp.h
//...

class ShapeImp {
      public:
            ShapeImp( int x,
                        int y );

      private:
            int _x;
            int _y;
}

//...
```



Insulation – Fully-insulating Concrete Class

```
//shape.h
//...

struct Shape_i;

class Shape {
      public:
            Shape( int x, int y );

      private:
            Shape_i* _data;
}

//...
```

```
//shape.c
//...

struct Shape_i {
      int x;
      int y;
}

//...
```



Conclusions

- Physical dependencies are root causes for poor quality in software design.
- Physical dependencies are implied by logical design.
- Acyclic design hierarchies should be preferred. Cyclic dependencies should be avoided.
- There exist techniques to reduce dependencies.

Thank you for your attention!