Genemodulabstraxibilicity

The feeling that it's *just too difficult*

Steve Love

steve@arventech.com

ACCU April 2010

Brief

Genemodulabstraxibilicity

Extensibility	Maintainability
Abstraction	Extensibility
Reusability	Adaptibility
Generality	Modularity
Testability	Genericity
Flexibility	Cohesion
Quality	Stability
Clarity	Utility

Simplicity

Part 1

Symptoms

1. Code Smells
   - The usual suspects
   - Clever tricks
   - Dependency jungle
2. Design Smells
   - One size fits all
   - Rigidity
   - The Monolith

The usual suspects

A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program. [...] Examples are the exclusion of goto-statements and of procedures with more than one output parameter.

Edsger Dijkstra, "The Humble Programmer", CACM 1972

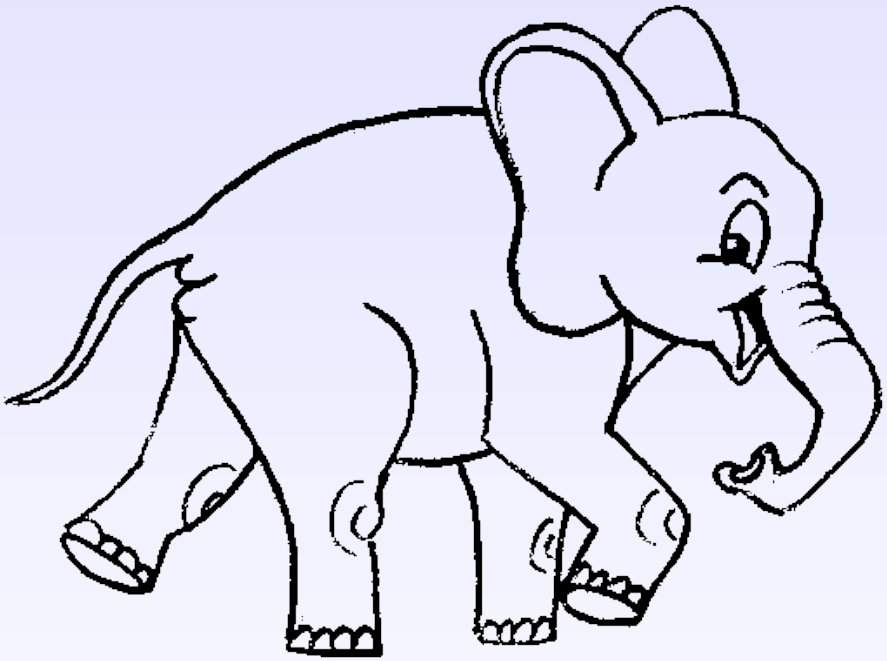
Incomprehensible flow

goto, break, continue....and throw?

non-local variables and singletons

multiple returns

looooooooooooooooong functions



Variations on a name

```
accumulator = x + 2;
accumulator2 = accumulator + y;
```

Often manifested in over-long functions.

Similar and related to variable re-use.

Variable re-use

Would this example be improved with separate start and end variables? :

```
DateTime stamp = DateTime.Now;
Console.WriteLine( "Start at {0}", stamp );

// ... Perhaps use stamp elsewhere

stamp = DateTime.Now;
Console.WriteLine( "Finish at {0}", stamp );
```

Dumb getters and setters

Making stuff private does not “cause” encapsulation.

Encapsulation is an effect, not a cause

Encapsulation should not be a design goal.
Good designs *achieve* encapsulation...
it doesn’t happen by accident.

A forest of comments

Example

```
# sum the elements of sequence seq
total = 0
for i in seq:
    total += i
```

Better

```
total = reduce( lambda x, y : x + y, seq )
```

A forest of comments

Example

```
# sum the elements of sequence seq
total = 0
for i in seq:
    total += i
```

Better

```
total = reduce( lambda x, y : x + y, seq )
```

Best

```
total = sum( seq )
```

Unused stuff

Is this slide pointless?

```
if( is_valid )
{
    result = x;
}
else
{
    result = x;
}
```

Clever tricks

It takes one of two different forms: one programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question "Can you code this in less symbols?" [...] or he just asks "Guess what it does!".

Edsger Dijkstra, “The Humble Programmer”, CACM 1972

Operator madness

```
class employee
{
public:
    // What does this mean?
    bool operator<( const employee &
        other ) { ??? }
};
```

Common motivations include being able to use such types in e.g. `std::set` in C++.

Operator madness

Counter examples:

```
string operator+( const string & l, const string & r );

template<class T>
boost::basic_format& operator%( const T & x )
```

Operator madness

Mis-uses may be subtle.

Example

```
datetime & datetime::operator--();
```

Is this by 1 second? 1 nanosecond? 1 day? Who knows?

Overloading overloading

Which one of these gets called?

```
void get_result( long l );
void get_result( void * p );
// ...
get_result( 0 );
```

Overloading overloading

Which one of these gets called?

```
void get_result( long l );
void get_result( void * p );
// ...
get_result( 0 );
```

Needing to be an expert on name lookup doesn't help!

Shiny and new

Whenever new features appear it’s tempting to try them everywhere...

...all nails and a shiny new hammer

Hieroglyphic code

```
string src = get_large_string();
string rep = "\\/";
char with = '.';

replace_if( src.begin(), src.end(),
   bind(
      not_equal_to< string::const_iterator >(),
      bind( find< string::const_iterator, char >,
         rep.begin(), rep.end(), _1 ),
      rep.end() ),
   with );
```

Don't Do That!

It's not big and it's not clever.

Stynamic typing

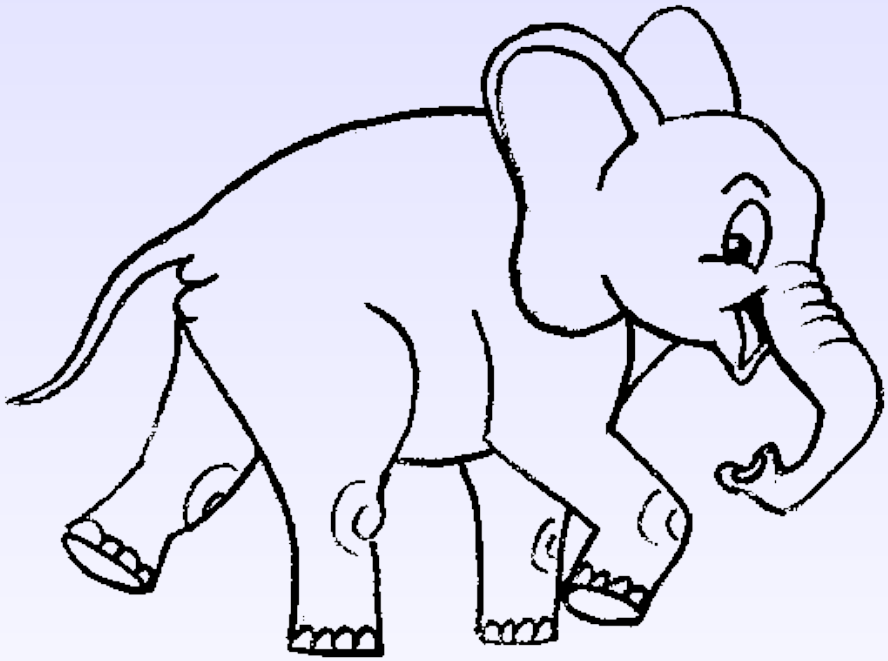
It’s hard to use a dynamic languages like Python as a statically-typed language, and it looks odd if you do this everywhere:

```
result = string( "foo" )
value = int( 10 )
```

So, don’t try to make your statically typed language like a dynamic one everywhere:

```
var host = "myhome.com";

auto name = "Arthur C. Clarke";
```



Old tricks, new dog

LINQ may be an excuse to write inline SQL.

```
var result = new SqlCommand(
    "SELECT id, name FROM products",
    connection ).ExecuteReader();
while( result.Read() ) { ... }
```

versus

```
var result =
    from product in products
        select product;
foreach( var prod in result ) { ... }
```

Dependency jungle

Short-term convenience
Long-term consequence

Inheritance trap

Fact

Public inheritance is the strongest form of coupling.

Spurious relationships often result in needless inheritance.

Inheritance trap

Fact

Deeply nested inheritance makes code obfuscated.

```
public class Option : Instrument
{
  public void Price()
  {
    // ...
   double rate = GetRate();
    // ...
  }
  // ...
}
```

What does GetRate call?

- Free function? Global or namespace?
- Member function? Which?:
  - This class?
  - Base class?
  - Derived class?
- Is the base class an interface?

Multiple versions

```
 Directory of G:\work\lib
05/04/2010  06:58    <DIR>          .
05/04/2010  06:58    <DIR>          ..
05/04/2010  06:58    <DIR>          boost_1_38_0
05/04/2010  06:58    <DIR>          boost_1_39_0
05/04/2010  06:58    <DIR>          boost_1_40_0
...
```

Size DOES matter

Go on a version diet!

One size fits all

The open secrets of good design practice include the importance of knowing what to keep whole, what to combine, what to separate, and what to throw away.

Kevlin Henney, "From Mechanism to Method: Distinctly Qualified", Dr Dobbs, May 2001

LongOrVagueFunctionNames

```
void
    ShowWaitCursorAndUpdateDatabaseWithProgressDisplayCl
    eanupTempDir();

void DoIt();
void TestMyClass();
```

These are (probably) functions with too many hats. The first one is a bit more honest about it!

Test functions with long descriptive names are different

```
void TestThatResizedSquareHasEqualWidthAndHeight();
```

“Complete” interfaces

If you haven’t got one of these, then you’ve missed something...

Speculative virtuals

...and protected data

```
public class summat
{
    public virtual int width()
    {
        get{ return w; }
        set{ w = value; }
    }

    protected int w;
}
```

Classes should be base classes *by design* not accident.

Rigidity

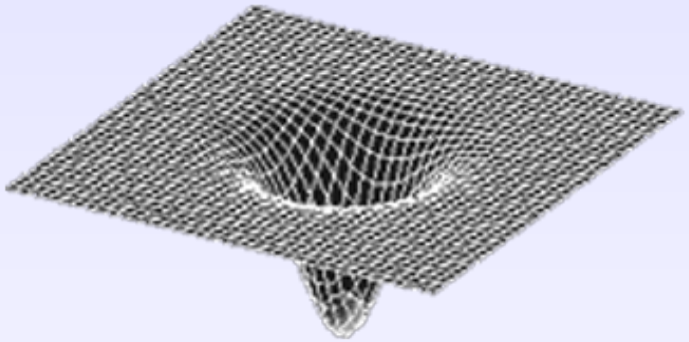
It is hard to change because every change affects too many other parts of the system.

Robert C. Martin, "The Dependency Inversion Principle", C++ Report, May 1996

Singleton and other globals

Beware the allure of global state

Singleton
Monostate
Global variables



Star

- Stealth coupling
- Dependency centre of gravity
- Lifetime and scope management

Concrete inheritance

Inheritance is a powerful tool for expressing your deisgn to the computer...

...and to other programmers!

A Circle is not an Ellipse

Use of concrete instead of abstract types leads to rigidity and fragility.

Concrete inheritance

Inheritance is a powerful tool for expressing your deisgn to the computer...

...and to other programmers!

A Circle is not an Ellipse

Use of concrete instead of abstract types leads to rigidity and fragility.

(**no matter what the mathematicians tell you - because they may also try to explain why a teacup is also a doughnut*).

The monolith

the four million-year-old black monolith has remained completely inert, its origin and purpose still a total mystery.

Arthur C. Clarke, 2001 A Space Odyssey

Include the world

Header files that include other header files.

...and then don't use them
...or could operate just as well
with forward references

Components that reference other concrete components.

...or worse,
abstract components that reference concrete ones

Unthinking references

Directly referring to a concrete type instead of an abstract version leads to rigidity and brittle code

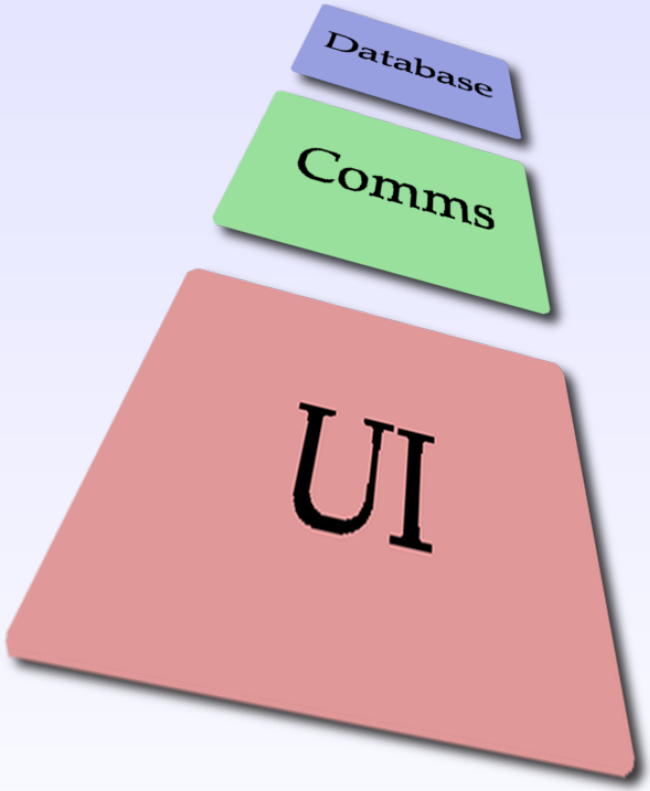
```
WifiComms comms = new WifiComms();
```

or

```
bool Attach( WifiComms comms )
{
}
```

Especially *if that type is in another module*

The dependency horizon



Dependencies are transitive: if A depends on B, and B depends on C, then A depends on C.

Part 2

Diagnoses

3. Responsibility
   - Too much
   - Too little
   - Indeterminate
4. Testing
   - Untestable
   - Testing considered useless
5. Requirements Gap
   - Feature angst
   - Creeping scope

Too much responsibility

...too many hats!

General

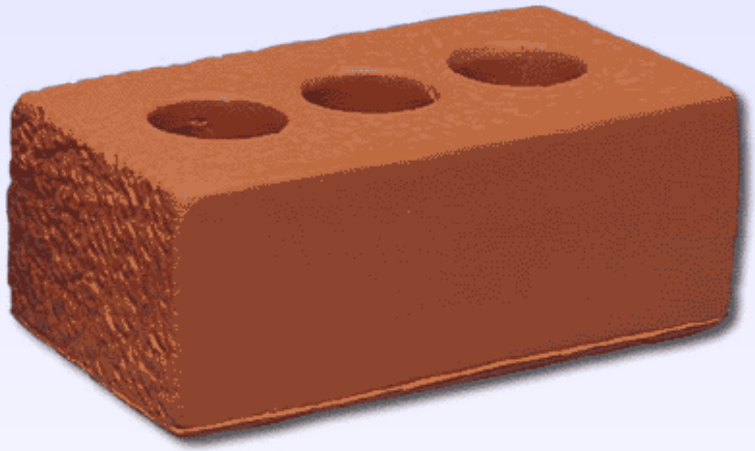
Featurism Take part in all manner of expressions, often for the sake of “cute” brevity, e.g. conversion operators

Speculation Provide various hooks for future extension, even if they make no sense to role of the class

Promiscuity Able to be used by other systems, e.g. STL containers and DI frameworks, regardless of purpose

Designed for re-use

Design for *use*, not *re*-use.



Predicting the future of code is just as hard as predicting the future of anything else...

Out parameters

- Functions that return more than one thing.
- Procedures that have more than one side-effect.

```
bool connect( string dbName, ref DbConn conn, ref List<
    string > errors )
```

- C++ non-const pointer vs. non-const reference.

Too little responsibility

While none of the work we do is very important, it is important that we do a great deal of it.

Joseph Heller, “Catch-22”, 1961

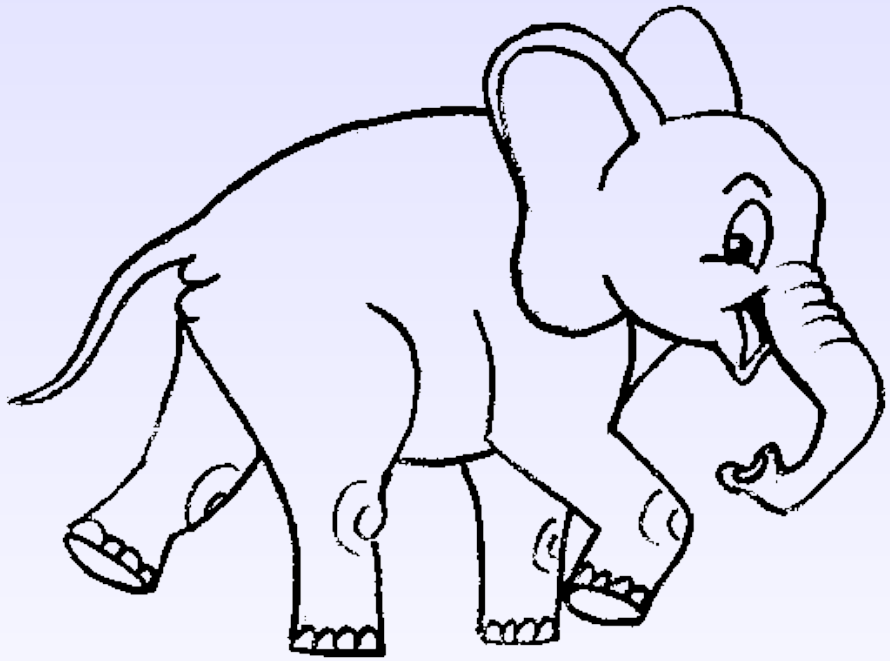
Simplisticity

All collaborators and no behaviour - just plumbing.

Accessors for every field.

Written in a “straight line”.

These are not necessarily *useless* bits of code, unless they make up the majority of your codebase...



Accidental relationships

Business logic spread out through lots of functions, classes, components, even technologies

Irresponsible code

- Resurrected after Disposal
- Clever and broken custom allocators
- Overriding the global new operator

Indefinite responsibility

Information Retrieval has got him down as inoperative. And there's another one - Security has got him down as excised. Administration's got him down as completed

From "Brazil", 1985, Terry Gilliam

No discernible role

Miscellaneous repositories:

```
#include <windows.h>

import java.util;
```

Indeterminate types:

```
LRESULT CALLBACK WndProc( HWND hwnd, UINT message,
    WPARAM wParam, LPARAM lParam )

System.Decoder.Convert( byte[], int, int, char[], int,
    bool, int, int, bool )
```

Obscured by plumbing

```
//...
try
{
    log.Write( "Begin calc" );
    #region critsec
    lock( lockObject )
    {
        results.Add( value );
    }
    log.Write( "End calc" );
    #endregion
}
catch( Exception x )
{
    lock( lockObject )
    {
        errors.Add( value );
    }
    log.Write( x.Message );
}
```

Untestable

Getting existing code under test can be....stressful.

Code can be hard to test for a variety of reasons....

No one thing to test

Unclear responsibilities

Muddy implementations

Collaborations of convenience

Behaviour by side-effect

Singletons and globals
Difficult to replace when testing
May bring unwanted dependencies
Lock-in implementation

Mutable collaborators
Can be replaced for testing
How much behaviour?
End up testing the (mocked) collaborator logic instead of the real thing...

Testing considered useless

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger Dijkstra, “Notes On Structured Programming”, 1970

One test to rule them all

"Yes, I've unit tested the code...."

```
[TestFixture]
public class TestPerson
{
  [TestCase]
  public void Run()
  {
    SetupData();
    Assert.Equals( person.Name, expectedName );
    Assert.Equals( person.Age, expectedAge );
    Assert.IsNull( person as Manager );
    Assert.IsNotNull( person as Employee )
    // ...
```

Micro tests

“My code has a huge set of unit tests....”

```
money.setExchangeRate( 2.34 );
assertTrue( money.getExchangeRate().equals( 2.34 ) );
money.setExchangeRate( 56.7 );
assertTrue( money.getExchangeRate().equals( 56.7 ) );
// ...
```

(...which aren’t really testing very much!)

Fragile tests

The numbers are wrong.

Tests must change when the *expected* numbers change.

What if the expected numbers are wrong?

Feature angst



Indecision

Choice of different...

API
Technology
Implementation possiblity

Don’t compensate for ambiguous or amorphous requirements by making code general enough to handle all the options. That path leads to madness...

Absence

A ***different*** kind of indecision.

...different **component**

...different **team**

...different **organisation**

Creeping scope

But within a one-hour lecture...he managed to ask for the addition of about fifty new features, little supposing that the main source of his problems could well be that it contained already far too many “features”.

Edsger Dijkstra, “The Humble Programmer”, CACM 1972

(About PL/1)

Confused class

The name of a thing -

class	function	variable
service	module	namespace

- is important

Know what a thing *IS* and you know what to call it.

Option explosion

- Conditional compilation
- Lots of branches
- Configurable *everything*

When scope changes, *that* is a hint to hide the detail in abstraction.

A short diversion

A story from history

The 1960s and 1970s were turbulent times for programmers.

The cost of software was out-stripping that of hardware.

Project failure was expensive and well-publicised.

This was called...

The software crisis

"The turning point was the Conference on Software Engineering in Garmisch, October 1968, a conference that created a sensation as there occurred the first open admission of the software crisis."

Edsger Dijkstra, "The Humble Programmer", CACM 1972

Here are some observations from the time....

Modularity

"[T]he quality of programmers is a decreasing function of the density of go to statements in the programs they produce."

"A Case Against The Go To *Statement", a.k.a* "go to *Statement Considered Harmful". 1968 (Edsger Dijkstra)*

Locality

“[T]he non-local variable is a major contributing factor in programs which are difficult to understand.”

“Global Variables Considered Harmful”. 1973, William Wulf and Mary Shaw

Abstraction

“As soon as people learn to apply principles of abstraction consciously, they won’t see the need for go to, and the issue will just fade away.”

“Structured Programming With Go To Statements”, 1974, Donald E. Knuth

Be a shame not to mention...

“Having at last put to rest to GOTO controversy, we now may enter the era of the COME FROM conundrum. “

R. Lawrence Clark, Datamation, 1973

The last 40 odd years

1968 “Software Crisis” term used at the NATO SW Conference

1986 “No Silver Bullet”, Fred Brooks and complexity in software

The last 40 odd years

1968 “Software Crisis” term used at the NATO SW Conference

1986 “No Silver Bullet”, Fred Brooks and complexity in software

1999 The Millennium Bug

The last 40 odd years

1968 “Software Crisis” term used at the NATO SW Conference

1986 “No Silver Bullet”, Fred Brooks and complexity in software

1999 The Millennium Bug

2000 It all still works! Yay! Back to business as usual

The last 40 odd years

1968	“Software Crisis” term used at the NATO SW Conference
1986	“No Silver Bullet”, Fred Brooks and complexity in software
1999	The Millennium Bug
2000	It all still works! Yay! Back to business as usual
Today	64 Bit, multi-core
Tomorrow	? Never mind what’ll happen in 2038...

Nothing new under the sun...

"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."

Edsger Dijkstra, "The Humble Programmer", CACM
1972

Treatments

6. Be responsible
   - Distribute responsibility
   - Design for test
7. Manage complexity
   - Be adaptible
   - Manage complexity
   - The old ones are the best ones

The oldest debate

Locality Modularity Abstraction

These are the core of our craft, and have been debated for 30, 40 even 50 years!

Simplicity

"Go to considered harmful" was never about goto itself - it's an essay about modularity and comprehendability.

> *"The price of reliability is the pursuit of the utmost simplicity."*

C.A.R. Hoare, "The Emporer's Old Clothes", CACM, 1981

Distribute responsibility

Many hands make light work

Proverb

Refactor

Long functions usually need splitting up

A short function calling lots of other smaller functions is better...

...but not ideal

Order out of chaos

```
void get_targets( map< string,vector< string > > & targets)
{
  for( auto i = targets.begin(); i != targets.end(); ++i ) {
    if( i->first == "URL" ) {
      for( auto j : i->second ) {
        if( ! j->find("http://") != 0 )
          throw exception( "Bad format" );
        urls.push_back( *j );
      }
    }
    else if( i->first == "LOCAL" ) {
      for( auto j : i->second ) {
        if( ! j->find("file:///") != 0 )
          throw exception( "Bad format" );
        files.push_back( *j );
      }
    }
  }
}
```

Order out of chaos

Improved a little by extracting a new role:

```
struct url
{
  url( string type, string name )
    : name( name )
  {
    if( type == "URL" && ! name.find( "http:") == 0 )
     throw exception( "Bad format" );
    else if( type == "LOCAL" && ! name.find( "file:" ) == 0 )
      throw exception( "Bad format" );
  }
  string name;
};
```

(Of course a *real* one would be much more sophisticated...)

Cohesion

A class' data members are non-local variables

use them wisely!

"m_" doesn't make a huge function more comprehendable.

and are pointless in short ones

Cohesion is about responsiblity and locality NOT data hiding.

Design for test

Truth is what stands the test of experience

Attributed to Albert Einstein

Good companions

Testable	Decoupled
Flexible	Easily adapted
Portable	No platform specific dependencies
General	Single role easily “re-used”

Be adaptible

As a matter of fact, the adaptability of a program to changes in its objectives (often called maintainability) and to challenges in its environment in terms of the degree to which it is neatly structured.

Niklaus Wirth, "Program Development by Stepwise Refinement", CACM 1971

Prepare for change

speculative generality != adaptibility

Change happens.
Flexibility in the face of change “happens” for cohesive, decoupled and simple code.

Indecision is positive

Henney's Uncertainty Principle

Use the uncertainty as a driver to determine where you can defer commitment to details and where you can partition and abstract to reduce the significance of design decisions.

Manage complexity

Gadgets and glitter prevail over fundamental concerns of safety and economy.

C.A.R. Hoare, “The Emporer’s Old Clothes”, CACM, 1981

Prefer simplicity

Simple code does not mean ignoring complex concepts.

Iterator concepts in C++
Anonymous C# delegates
Python's list comprehensions
Polymorphic behaviour in any OO language

Ignoring such things may even result in *more* complex code!

Occam's razor

Simplistic != Simple

Choose the simpler of two ***equivalent*** things.

Distinguish between accidental and essential complexity.

Literacy

New features are introduced (usually) for good reason.

Know those reasons, and when to use the feaure *effectively*.

```
auto record = make_tuple( x, y, z );
```

Be specific

Abstraction is selective ignorance

not an excuse to be vague

The old ones are the best ones

...of course, there's a reason for that...

Dependency inversion

1. High level modules should not depend upon low level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions.

Robert C. Martin, "The Dependency Inversion Principle", C++ Report, May 1996

Single responsibility

The Single Responsibility Principle

There should never be more than one reason for a class to change

Robert C. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice Hall 2002

All in all...

Genemodulabstraxibilicity

- Generality
- Modularity
- Abstraction
- Flexibility
- Reusability
- Simplicity

Sometimes these are competing attributes

Don’t try to do too much all in one place

If you feel it’s just too difficult...

If you feel it’s just too difficult...

Don’t Panic!

A good, healthy, sceptical look can show you where to begin....

Thanks!