

Bubbles

Producing efficient and reliable code, first time, every time.

by Chad Z. Hower a.k.a. Kudzu

Table of Contents

Part I Introduction	3
Part II Language Independent	3
Part III Philosophy, not a Religion	3
Part IV Bubbles and Boxes	4
Part V Bubelen	4
Part VI Box Testing	4
1 What is Box Testing?	5
2 Box Colors	5
Black Box	5
White Box (a.k.a. Grey Box)	5
Glass Box	5
Pink Polka Dot Box	5
Tinder Box	6
3 Examples	6
Black Box Example	6
Glass Box Example	8
Tinder Box	8
Likely Scenario.....	9
Random Scenario.....	11
Part VII Preparations	11
Part VIII Goals	12
1 Regression Testing	12
2 Test First	12
3 Bug Verification	12
4 Stress Testing	13
5 Delegation	13
6 Debugging	13
7 Playgrounds	13
8 Team	14
9 Pair Programming	14
10 Profiling	14
11 Memory Leak Detection	14
Part IX Features	14
1 User Data	14

2 Parameters	14
3 External Data	15
4 Visual Feedback	15
5 Visual Interface	15
Part X Conclusion	15
Part XI Produced with Help and Manual	15
Part XII About the Author	15
Index	0

1 Introduction

Producing reliable code requires continuous testing, which is something many programmers dread. Because of this testing is often cut short, or in some cases completely omitted. This essentially leaves testing to the end users.

Bubbles are designed to make testing easy, and even attractive for developers to use.

A bubble is a piece of code for testing, developing, profiling, debugging, and stress testing. Bubbles are similar to boxes which are used in box testing (sometimes also referred to as unit testing), however bubbles are not designed with the only goal being testing.

The process of creating bubbles is referred to as bubbling, and the default utility provided as open source is called Bubelen. Bubelen through its use of bubbles, uses a different philosophy than other frameworks.

Bubbling is more than just testing, or a process that is applied after development has occurred. Bubbling can be applied to existing development, but the philosophy of bubbling is to use bubbling from the start.

When used from the start, no application or end product is built until everything has been built in bubbles first. Each bubble tests functions, components, classes or other units of code. These are referred to as blocks. Bubbles are used to test blocks, but it is not necessarily a one to one relationship. Many bubbles may exist to test a block, or combination of blocks.

Only after that, are the blocks combined into a final application, or component set (target). For on going development, blocks are tested using bubbles before being used in the target. The use of bubbling also enforces a good development design of componentization, encapsulation, and if a user interface exists, keeping a separation between it and the program logic.

This is my "first draft" at documenting a process that I have been using for quite some time. I will be adding more to this article as time permits.

2 Language Independent

Bubbling is language independent and is simply a philosophy of best practices and development methods. The reference implementation is written in Delphi, but bubbling itself is not tied to Delphi or any other language.

This paper focuses on the philosophy and applications of bubbles. This paper does not describe any implementations or make reference to specific languages, with the exception of the overview of box testing.

3 Philosophy, not a Religion

Bubbling is a philosophy, not a religion.

This means that bubbling is compatible and sometimes complementary to other philosophies. Two such philosophies that are complementary to bubbling are Extreme Programming (XP) and Refactoring. Bubbling complements these as each addresses a different issue. XP focuses on project lifecycle, refactoring on improvement of code, and bubbling on quality awareness.

Like many philosophies, the authors were not necessarily the inventors, but just the first to document,

refine, and provide a teaching point. I have been using XP and refactoring for many years, long before they became documented philosophies. I suspect many others were too, and the same may be said of bubbling itself.

This also means that bubbling is voluntary. If you like what bubbling offers, you study it and learn from it.

Bubbling is at odds with a with a few philosophies however. If you are a disciple of such cults as the Zen of Goto, or the Sect of Cuttenpaste, you may find bubbling uncomfortable.

4 Bubbles and Boxes

Bubbles in some aspects may appear similar to boxes. That is because they are. Bubbles encompass everything boxes do, but also much more. Bubbles are also constructed in similar manners.

Bubbles are different than boxes in that they are designed for more than just simple pass/fail unit testing. Bubbles not only support more advanced forms of testing, but support playgrounds, visual feedback, historical data, profiling, and more.

Some might ask - why not just extend boxes? That was my first idea as well. But when I proposed such ideas within QA forums relating to extensions for existing box testing frameworks, I was sternly dismissed and told that my ideas were "incompatible" and not the purpose of boxes. It was recommended that I use boxes, plus a memory leak tool, plus a profiler, plus custom test utilities, etc.

This is quite cumbersome to run various independent tools to accomplish what can be done as an integrated process. Thus, the birth of bubbles. "Think outside of the box".

5 Bubelen

verb, from middle english, to bubble

Bubelen is an open source implementation of a system for building and maintaining bubbles written in Delphi. Bubelen grew out of Boxster, which was a box testing system I devised in 1997. Boxster was used by the Indy Pit Crew and several others during its life.

Bubelen is by no means the only system that can be used for bubbles, it is just the reference system. Bubelen is available at <http://www.indyproject.org/>.

6 Box Testing

Since bubbles have some similarities with boxes, boxes will be covered briefly. The terminology and information applies to the unit testing part of bubbles.

When I first started into researching box testing many years ago, I researched black boxes. However, I soon discovered that there were several other box colors. These included white and glass boxes, in addition to black boxes. I have also added one of my own. It is appropriately named the "tinder box" technique.

6.1 What is Box Testing?

For box testing, first it must be defined what a "box" is. A box is a defined piece of code that you wish to test.

Boxes are then created, and tests are performed against the boxes. When any particular test fails against a box, the problem is easily pinpointed because it is known what is in the box. By defining boxes, it will always be known which boxes to test when code has been changed.

After your boxes are defined, the next step is to try to break them. Exactly how to break them is determined by the boxes color.

6.2 Box Colors

Boxes are named by using colors. There are many different colors of boxes.

6.2.1 Black Box

To construct black boxes, you need to know what a functions inputs and correlating outputs are. You then create a series of calls that call the function with various inputs, and test their outputs. With black boxes, you can not see what is happening while the function is in progress, nor can you count on any knowledge of how the function operates. The only two items of knowledge that you can rely on are its inputs, and its outputs.

Black box testing also targets end-point functions. End-point functions are functions that do not rely on other functions. When black box testing is used on non end-point functions, you must also separately black box test the functions that it calls. Because of this, complete black box testing can be quite involved.

6.2.2 White Box (a.k.a. Grey Box)

I was not able to find a definitive definition of a white box, but I believe I understand the overall definition. White boxes are very similar to black boxes, with the exception that non-endpoint functions are targeted. If the tested function passes, functions that are called from within it are considered valid.

The reasoning behind white boxes appears to be that if functions are only called from other functions, and those functions perform properly under those conditions, it does not matter if they fail under conditions that do not occur.

6.2.3 Glass Box

Glass boxes are similar to black boxes as well. However, with glass boxes, knowledge of the internals of the functions is available. Some references indicate that glass boxes and white boxes are identical, but others specify that they are different from black boxes.

For this article, glass boxes will be kept separate from black boxes. Knowing the internals of a function, a flow diagram can be established. Using the flow diagram the box can completely cycle through inputs that trigger all internal paths of the function guaranteeing that all parts of the function are executed and tested.

6.2.4 Pink Polka Dot Box

There is no such thing as a pink polka dot box. However you will be tested on this question later, to see if you were paying attention.

6.2.5 Tinder Box

Tinder box is a box type that encapsulates a personal technique that I have used for many years. Many functions work properly when no exceptions occur. However, when exceptions occur, the functions do not properly handle the exceptions and often cause other side effects. In other cases the functions handle the exception, but not in the manner that they were designed to. Using the tinder box technique, exceptions are purposefully introduced into functions. The goal of tinder box testing is to intentionally and purposefully blow the functions up.

6.3 Examples

The following examples will demonstrate various usage of various box types to test a sample function. Delphi has a built in function in SysUtils called StringReplace. A modified form of this function will be used in this example. This modified version is called StringThing.

The example included here is very simplistic and is probably not a good example of a real world situation. It should serve however as a basic introduction. In the future I plan to add a more in depth example. Until that time, the test suite for Indy is a very good resource and is available in the Indy Development Snapshot.

Here is the documentation for StringThing (The function has been modified and extended to better fit the example):

StringThing returns a string with the occurrences of one substring replaced by another substring.

```
type
  THandleFlags = set of (rfReplaceAll, rfIgnoreCase);

function StringThing(const S, OldPattern, NewPattern: string; Flags:
  THandleFlags): string;
```

Description

StringThing replaces occurrences of one substring, specified by OldPattern, with another substring, specified by NewPattern, in a given string. StringThing assumes the string may contain Multibyte characters.

If the Flags parameter does not include rfReplaceAll, StringThing only replaces the first occurrence of OldPattern in the string S. If the Flags parameter includes rfIgnoreCase, The string comparison operation is case insensitive.

OldPattern must not be a null string, if so, EEmptyString exception will be raised.

6.3.1 Black Box Example

Using the documentation for StringThing, a black box can be constructed to test StringThing. The inputs and the associated outputs for StringThing can be determined from the documentation. For each function testes, a harness will be created that uses the function name, prefixed by "Verify". This function will call the test function, and return parameter info as a string if the test fails.

```
function VerifyStringThing(const s, OldPattern, NewPattern: String;
  Flags: THandleFlags; EClass: ExceptionClass; ExpectedResult: String): String;
var
  LParams, LResult: String;
begin
  Result := '';
  LResult := '';
  LParams := 'StringThing' + #13#10
    + ' S: ' + s + #13#10
    + ' OldPattern: ' + OldPattern + #13#10
```

```

+ ' NewPattern: ' + NewPattern + #13#10;
  try
    LResult := StringThing(s, OldPattern, NewPattern, Flags);
  except
    on E: Exception do begin
      if E.ClassType <> EClass then begin
        Result := 'EXCEPTION: ' + E.Message + #13#10
      end else begin
        EClass := nil;
      end;
    end;
  end;
end;
if (LResult <> ExpectedResult) or ((EClass <> nil) and (Result = ''))
or (Result <> '') then begin
  Result := Result + LParams + ' Result: ' + LResult + #13#10;
  if EClass <> nil then begin
    Result := Result + 'Expected Exception: ' + EClass.Classname
    + #13#10;
  end;
  Result := '-----' + #13#10 + Result;
end;
end;

```

Another function is then created that attempts to stress the function by calling it with different inputs and comparing the outputs to expected results. An example appears next.

```

procedure TFormMain.FormShow(Sender: TObject);
var
  s: string;
  LTest: string;
begin
  s := '';
  LTest := 'The quick brown fox is quick';

  // Test by replacing only the first occurrence
  s := s + VerifyStringThing(LTest, 'quick', 'slow', [],
    , nil, 'The slow brown fox is quick');

  // Test by replacing all occurrences
  s := s + VerifyStringThing(LTest, 'quick', 'slow', [rfReplaceAll]
    , nil, 'The slow brown fox is slow');

  // Test that case sensitivity does work
  s := s + VerifyStringThing(LTest, 'Quick', 'slow', [rfReplaceAll]
    , nil, 'The quick brown fox is quick');

  // Test that case insensitivity does work
  s := s + VerifyStringThing(LTest, 'Quick', 'slow', [rfIgnoreCase]
    , nil, 'The slow brown fox is quick');

  // Test that exception is thrown when oldpattern is an empty string
  s := s + VerifyStringThing(LTest, '', 'slow', [], EEmptyString
    , '');

  if Length(s) = 0 then begin
    s := 'All Tests Successful';
  end;
  Clipboard.AsText := s;
  Listbox1.Items.Text := s;
end;

```

The example tests the function by calling it with a variety of parameter combinations. This is done to stress the tested function as much as possible. Not all parameter combinations have been tested in this example, but to properly stress the function they should be. In many cases, a loop can be used to stress the function. In others, parameters and expected results can be stored in an array or a file. If a file is used, it facilitates easy by a QA or testing department, as they are free to change the parameters on their own.

This example is an example of a black box. The box is based solely on knowledge of inputs and how

the outputs should correspond, without any knowledge of how the function is implemented internally. In short, the box have verifies that the function works as the documentation says. Black box testing is useful on your own source code, but it is especially useful for libraries that source code is not available.

This example is available as StringThingBlackBox.dpr.

6.3.2 Glass Box Example

The next example will describe how to test StringThing using a glass box. To use a glass box the implementation of the function needs to be known and understood. The StringThing function is listed next.

```
function StringThing(const S, OldPattern, NewPattern: string;
  Flags: TReplaceFlags): string;
var
  SearchStr, Patt, NewStr: string;
  Offset: Integer;
begin
  if length(OldPattern) = 0 then begin
    raise EEmptyString.Create('Old Pattern must not be empty.');
```

```
  end;
  if rfIgnoreCase in Flags then begin
    SearchStr := AnsiUpperCase(S);
    Patt := AnsiUpperCase(OldPattern);
  end else begin
    SearchStr := S;
    Patt := OldPattern;
  end;
  NewStr := S;
  Result := '';
  while SearchStr <> '' do begin
    Offset := AnsiPos(Patt, SearchStr);
    if Offset = 0 then begin
      Result := Result + NewStr;
      Break;
    end;
    Result := Result + Copy(NewStr, 1, Offset - 1) + NewPattern;
    NewStr := Copy(NewStr, Offset + Length(OldPattern), MaxInt);
    if not (rfReplaceAll in Flags) then begin
      Result := Result + NewStr;
      Break;
    end;
    SearchStr := Copy(SearchStr, Offset + Length(Patt), MaxInt);
  end;
end;
```

With larger functions, a flow diagram should be created. Next, ensure that the test harness causes the function to execute each path at least once. The only variance in this function is its if statement.

Reexamining the black box test will show that it properly tested both branches. In this case, the glass box would be identical to the black box. In a situation like this, where glass box testing does not expand your black box model, it speaks well of your original test pattern. You may not always be able to achieve this however, especially in larger and more complex functions.

6.3.3 Tinder Box

Tinder box testing is a simple technique. Every program should have exception handling. Of course, we all know how we think our functions handle exceptions. Unless something goes wrong though, how do you know if it will really handle the exception properly?

There are two types of scenarios with tinder boxes:

1. Likely - The introduction of errors likely to occur, at probable locations.

2. Random - The introduction of random errors, at random locations.

There are also two methods used to trigger (ignite) the tinder box:

1. Actual Errors - Actual errors can be introduced into the code by changing values of variables, setting pointers to nil, etc.
2. Explicit Exceptions - Explicit exceptions can be raised using raise.

6.3.3.1 Likely Scenario

The likely scenario tests code to see how it performs under slightly stressed, but not necessarily abnormal conditions.

Exceptions such as out of disk space, file is read only, file does not exist, and others of this type are inserted into locations that they are likely to occur. Breakpoints are then set on the line that the exception will be raised. The source code is then traced from that point to determine if it works as expected.

For the next example (available as WholeFileTinderBox\Step 1), a new function named WholeFile will be used. WholeFile, accepts a filename and returns the contents of the file in the result as a string.

```

unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFormMain = class(TForm)
    Memo1: TMemo;
    procedure FormShow(Sender: TObject);
  private
  public
  end;

var
  formMain: TFormMain;

implementation
{$R *.DFM}

function WholeFile(const AFile: string): string;
begin
  Result := '';
  if FileExists(AFile) then begin
    with TFileStream.Create(AFile, fmOpenRead) do begin
      if Size > 0 then begin
        SetLength(Result, Size);
        ReadBuffer(Result[1], Size);
      end;
      Free;
    end;
  end;
end;

procedure TFormMain.FormShow(Sender: TObject);
begin
  Memo1.Lines.Text := WholeFile(ExtractFilePath(Application.EXENAME) + 'Government.txt');
end;

end.

```

This is an easy example. You may spot the problem immediately, but let's use the tinder box to surface it. Assume something happens after the stream is created, such as a read error, out of memory, or any

other error. In this example, which exception occurs does not matter, so Exception itself will be used. If exception handling with try..except is also being tested, raising specific exceptions may be necessary.

For this test, an exception is inserted immediately after the TFileStream.Create to simulate an error occurring while opening the file. The code has one new line inserted, and it is highlighted in the following listing (available as WholeFileTinderBox\Step 2).

```

unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFormMain = class(TForm)
    Memo1: TMemo;
    procedure FormShow(Sender: TObject);
  private
  public
  end;

var
  formMain: TFormMain;

implementation
{$R *.DFM}

function WholeFile(const AFile: string): string;
begin
  Result := '';
  if FileExists(AFile) then begin
    with TFileStream.Create(AFile, fmOpenRead) do begin
      raise Exception.Create('Boom!');
      if Size > 0 then begin
        SetLength(Result, Size);
        ReadBuffer(Result[1], Size);
      end;
      Free;
    end;
  end;
end;

procedure TFormMain.FormShow(Sender: TObject);
begin
  Memo1.Lines.Text := WholeFile(ExtractFilePath(Application.EXENAME) + 'Government.txt');
end;

end.

```

Now set a break point on the "raise Exception.Create('Boom!');" line and run the application. Once the breakpoint is hit, step through the function until it exits. You should notice that the call to Free was skipped, and the function has a potential memory leak. To fix the leak, change the code as follows (available as WholeFileTinderBox\Step 3).

```

unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls;

type
  TFormMain = class(TForm)
    Memo1: TMemo;
    procedure FormShow(Sender: TObject);
  private

```

```

public
  end;

  var
    formMain: TFormMain;

  implementation
    {$R *.DFM}

  function WholeFile(const AFile: string): string;
  begin
    Result := '';
    if FileExists(AFile) then begin
      with TFileStream.Create(AFile, fmOpenRead) do try
        raise Exception.Create('Boom!');
        if Size > 0 then begin
          SetLength(Result, Size);
          ReadBuffer(Result[1], Size);
        end;
      finally Free; end;
    end;
  end;

  procedure TFormMain.FormShow(Sender: TObject);
  begin
    Memol.Lines.Text := WholeFile(ExtractFilePath(Application.EXENAME) + 'Government.txt');
  end;

end.

```

Now execute the code again, and trace from the exception. You will notice that the Free is now called, even when an exception occurs.

6.3.3.2 Random Scenario

The random scenario is used less often than the likely scenario, but it can really help to bulletproof your code. With the random scenario, a random exception is placed at a random location. Set a break point on the exception and execute the code. Then follow the code from the exception, and see how it behaves. I think you will be surprised how much of your code handles exceptions improperly or at least differently than you expect. Tinder box testing can be quite a humbling experience.

This is quite an elementary example. Tinder box testing becomes most valuable when you have existing exception handling in place and you need to test those handlers. Tinder box testing is even more valuable when the exception handling is nested or inter function. It is also very valuable in learning how try..finally and try..except blocks work if you are not familiar with them. I have found that the majority of programmers do not use proper exception handling, because they do not understand how it works completely. Tinder box testing is your chance to remedy this if you fall into this category.

7 Preparations

To facilitate bubbling, your code should contain certain characteristics. These characteristics are characteristics of good programming anyway, and possibly are already present in your code.

1. Properly documented procedure inputs and outputs.
2. Proper encapsulation as objects and functions.
3. Separated logic and user interface. This is often not the case with Delphi/C++ Builder projects, as logic is often embedded directly into events.

Do not panic if your code does not conform to item #3, some of my source code does not either. Bubbling can still be useful.

Bubbling is especially useful for component sets, as they inherently possess the above characteristics. However do not let this lead you to believe it is only for component sets. Bubbling should be used for all sorts of code including applications with user interfaces.

8 Goals

To better illustrate bubbling, its goals and applications, various items will be discussed next.

8.1 Regression Testing

Regression testing consists of retesting existing parts of a system after parts, or dependent parts have been altered intentionally or unintentionally. Regression testing is the cornerstone of box / unit testing.

Programmers love to tinker with code, often rewriting code to improve or optimize it. This often happens after the code is in use by many projects. Often any little change in functionality can have a snowball effect on other code, producing unexpected and disastrous results, usually in the form of bugs. Worse yet, they often do not appear immediately, making them harder to find, as they seem to appear from nowhere.

Regression testing allows software to be tested prior to each release as part of the QA (Quality Assurance) cycle. However regression testing should also be performed by developers, and very frequently. Performing regression testing frequently will catch bugs and other issues earlier and make them easier to isolate and fix.

Regression testing is normally applied to only logic. In system where logic is separated from the user interface, this allows for easy testing. However bubbles can also be used to test user interfaces, either separately, or in systems which the logic is embedded in the user interface. When the logic and interface are integrated, they must be tested as one unit and the results are not as definitive.

8.2 Test First

Test first is the practices of developing harnesses that "use" code, before the code actually exists. Code is then developed to fit this harness. The code is said to be complete when the harness functions fully.

In short, test first is building a box test before the code exists and developing the code to complete the box test.

In bubbling, the bubbles perform the function of the harness. By producing bubbles and using test first during development, development is more reliable and the master collection of bubbles is built up in an incremental manner for regression testing.

8.3 Bug Verification

Bugs often are reported, fixed, and then creep back into the code in later releases. Such bugs I call roaches. Roaches are often a sign of a weakness in a particular section of code. Because of the ability to recur, and the sign of a weakness it is very important to keep an active "roach alert". The best way to do this is to treat all bugs as potential roaches.

Typically a bug cycle is as follows:

1. Bug is reported by a user, tester, or developer.

2. Bug is verified, and if valid, recorded as a bug.
3. Bug is scheduled for fixing
4. Developer fixes bug.
5. Sometimes the developer tests the fix.

With bubbling the bug cycle is as follows:

1. Bug is reported by a user, tester, or developer.
2. Bug is verified, and if valid, recorded as a bug.
3. Bug is scheduled for fixing
4. A bubble is created to reproduce the bug.
5. Developer fixes bug by using the bubble.
6. Bubble is added to master collection of bubbles for regression testing.

This not only ensures that the bug has been fixed, but that it does not become a roach.

8.4 Stress Testing

Bubbling supports stress testing by using bubbles to subject blocks to excessive situations on a regular basis.

8.5 Delegation

Developers are often reluctant to write specifications documents and this weakness often extends to team leaders. Specs even when written are often lacking and ambiguous. It also becomes very tedious to write a document based specification for each small task.

Because of these factors, developers often implement task assigned to them that perform differently than intended by team leaders. This creates delays, bugs and unmet expectations.

With bubbling, test first is extended and team leaders create bubbles with descriptions when assigning tasks to developers. Developers then develop code to match the bubble.

In another variant, developers create the bubbles, but team leaders review them prior, and during implementation of the task. Team leaders review the bubbles to see if the implementation conforms to the intended vision.

8.6 Debugging

Debugging is greatly simplified with bubbling. In an optimal case, a bubble has found a flaw and is ready for use. In other cases a bubble must be built to reproduce a flaw. In such cases no time is wasted manually reproducing the conditions needed to debug.

8.7 Playgrounds

A key part of debugging in bubbling is the use of playgrounds. A contract (box) is enough to determine if a block is functioning or not, but is not enough to "see what is going on". Playgrounds offer this by allowing for visual and dynamic feedback based on input.

Playgrounds are specialized user interfaces integrated with a bubble for exactly this purpose. Playgrounds are extremely important during implementation and testing of complex blocks, or bubbles that cover compound blocks.

8.8 Team

Bubbling is conducive to team programming by enforcing encapsulation, testing, and by providing implementations of blocks in bubbles.

8.9 Pair Programming

Pair programming is the process of developing code partly and then handing off the code in mid process to another developer because of time constraints, need of sleep, time zone differences, or other. Normally pair programming requires very close communication and an overlap of working times, or close proximity to do so.

With bubbling, a bubble can easily be handed off quickly to another developer with only minimal introduction.

8.10 Profiling

Profiling is normally performed by language specific, often invasive tools that profile physical functions in source. The profiling is invasive and performed in a brute force manner. Such techniques are good for finding bottlenecks, but remedying can be difficult and diagnosis is often too late.

Bubbling supports profiling by measuring bubble execution times. This flags potential bottlenecks and performance problems during the development cycle.

By keeping historical data, and comparing against previous metrics, unintended detrimental changes can also be flagged during regression testing.

8.11 Memory Leak Detection

Memory leaks can be detected by checking bubbles before and after executions. This will not pinpoint the exact source of the problem as a full powered leak detector will, but it will serve as a flag that a more extensive leak detection tool should be used and a general area to look.

9 Features

Bubbling implementations must implement certain features essential to meeting the uses.

9.1 User Data

Bubbles support user specific data for conditions that may vary between developers.

Consider a bubble that implements an SMTP mail send. SMTP servers are private, and if hard coded only the developers who have access to that server will be able to successfully use the bubble. By using user data, the user is aware that certain parameters must be defined based on the local configuration.

9.2 Parameters

For regression testing, parameters are typically fixed. However bubbles support parameters for use in debugging and stress testing.

9.3 External Data

Testing often relies purely on tests built in code with hard coded values. This requires that code be modified to expand test conditions. Bubbles extend this to include external data.

External data can be used to easily test a wider variety of conditions for functions which work on larger pieces of data, or produce many outputs.

Consider a function which extracts HREF links from an HTML file. Instead of hard coding such tests into a bubble, a bubble uses external data specific to the bubble to obtain a list of external files. Each input file is passed to the bubble for processing by the block. The output of the block is then compared to a verification file associated with the input file.'

9.4 Visual Feedback

Box tests typically have little visual feedback aside from status of success or failure, and simple log data. Bubbles are more complex and often rely on visual feedback. Visual feedback is used to provide progress info for longer operations, and provide visual information useful to debugging.

9.5 Visual Interface

Box tests do not support visual interfaces unless separately implemented, and this is not part of the box testing philosophy.

Visual interfaces are necessary for implementations of playgrounds.

10 Conclusion

Bubbling if used will significantly reduce your development time and debugging time. Bubbling will help you to produce more reliable, more efficient. Bubbling will even find bugs that you otherwise would not find until reported by an end user. Furthermore, when bubbling does find a problem, you know exactly where it exists, and what conditions cause it to occur. Bubbling cannot find all bugs, but it can verify the absence of bugs.

Bubbling does require some investment of your time up front. In the end, you will find that the pay off is quite significant. The pay off is not only in terms of timesavings to you, but in happier customers, shorter testing and QA cycles, and overall reduced costs to your company.

11 Produced with Help and Manual

This document was produced with Help and Manual from ECSoftware. Help and manual is available at <http://www.helpandmanual.com>.

12 About the Author

Chad Z. Hower, a.k.a. "Kudzu" is the original author and project coordinator for Internet Direct (Indy). Indy consists of over 110 components and is included as a part of Delphi, Kylix and C++ Builder. Chad's background includes work in the employment, security, chemical, energy, trading, telecommunications, wireless, and insurance industries. Chad's area of speciality is TCP/IP networking

and programming, inter-process communication, distributed computing, Internet protocols, and object-oriented programming. When not programming, he likes to cycle, kayak, hike, downhill ski, drive, and do just about anything outdoors. Chad, whose motto is *"Programming is an art form that fights back"*, also posts free articles, programs, utilities and other oddities at Kudzu World at <http://www.Hower.org/Kudzu/>. Chad is an American ex-patriate who currently spends his summers in St. Petersburg, Russia and his winters in Limassol, Cyprus. Chad can be reached at cpub@Hower.org.

Chad works as a Senior Developer for Atozed Computer Software Ltd.
(<http://www.atozedsoftware.com/>)