IfSQ Level-1

An Entry-Level Standard for Computer Program Source Code

Second Edition
August 2008

Graham Bolton
Stuart Johnston

Contents

1. Management Overview

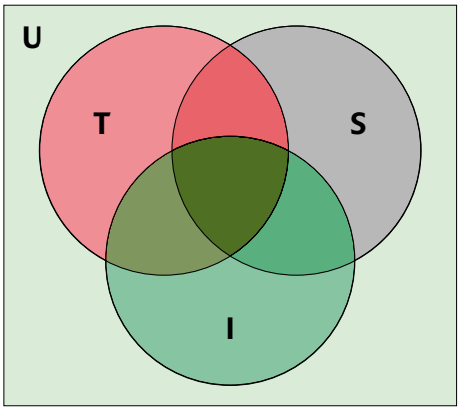
This booklet is about how to improve the quality of software through the use of code inspection.

Finding Software Defects

There are three ways of finding defects in any computer program:

- By **testing** the program, which involves running the program with a variety of input data and in a variety of scenarios to try and expose all error conditions. This is also referred to as dynamic analysis.
- By performing automated **static analysis**, in which the program is not run, but an analysis tool is used to process either the source or object code to flag possible coding errors.
- By **code inspection**, where a programmer visually examines the source code, looking for indications of poor programming practices or faulty logic.

As shown in Figure 1, these methods are complementary, and each is a useful component of the quality assurance process.



U: *All defects in a computer program*
T: *Defects found by Testing*
S: *Defects found by Static Analysis*
I: *Defects found by Code Inspection*

Figure 1. Coverage of software defect detection methods

IfSQ Level-1

Code Inspection

Code inspection has been proved to be at least 10 times more cost-effective than the other two methods. Despite this, it is the least-used, due to its human-intensive nature, and its reputation as being a tedious activity. However, neglecting this aspect of the quality process during software development greatly increases the risk of failure in a later stage.

To facilitate the acceptance of code inspections as a fundamental part of the development process, IfSQ proposes the adoption of a light-weight quality process, consisting of a three-level set of coding standards combined with an assessment method.

IfSQ has been applying this quality process for several years in projects of all sizes in the industrial, financial and governmental sectors. The end result is:

- Fewer defects
- Increased reliability
- Increased maintainability
- Increased control over development costs and suppliers
- Enhanced IT audit scope

Audience

This booklet is aimed at anyone involved in the software development process, including:

- Developers
- Testers
- Project managers
- Accountants
- IT Auditors
- Clients

2. Background

The Human Factor

In recent years the tools used by the software industry have become increasingly sophisticated and powerful. The use of modern modeling tools, integrated development environments, source control systems, code libraries, and automated test suites have freed today's architects, programmers, and testers from many of the repetitive aspects of their jobs, and made them more productive and efficient than their predecessors.

However as software consultant and author Gerald Weinberg wrote in *The Psychology of Computer Programming*, "Programming is first and foremost a human activity and only secondly something that involves computers." Human beings inevitably make mistakes, regardless of the technology supporting them. Indeed, overconfidence in the abilities of tools can make it easy to neglect quality practices fundamental to the craft of programming.

History's Worst Software Bugs

Simson Garfinkel 11.08.05

Sixty years later, computer bugs are still with us, and show no sign of going extinct. As the line between software and hardware blurs, coding errors are increasingly playing tricks on our daily lives. Bugs don't just inhabit our operating systems and applications -- today they lurk within our cell phones and our pacemakers, our power plants and medical equipment. And now, in our cars.

We see the results on a regular basis in the media, with stories of system failures, project overruns, and financial loss. Many of these cases have been traced back to simple errors resulting from the lack of a systematic approach to quality during system development.

Testing Is Not Enough

It is taken for granted that all software undergoes extensive testing before being released for use. But clearly, given the types of problems mentioned above, the test process alone is not sufficient to ensure bug-free applications.

While powerful test tools can hugely aid our ability to verify the correct working of programs, they cannot find every defect. All too often, bugs end up being discovered by the end-user. Any such defects caught during testing or production must go back to the programmer to be fixed, a far more time-consuming and expensive proposition than fixing them during the coding phase. Finding a defect by testing has been shown to be 10 times more expensive than finding a defect by inspecting the source code.[1]

A Lightweight Quality Process

In the last few years, within the software industry there have been attempts to move from the types of formal development methodologies commonly used in the 1980s and 1990s to a lighter and quicker set of processes. These so-called "agile" methods have proved to be popular and effective precisely because they are easy to comprehend and put into practice.

IfSQ strongly believes that what the software industry needs now is a set of quality standards, analogous to the agile development methods, that are easy to understand and apply. More importantly, IfSQ believes that reducing the threshold to achieving code consistency and completeness will encourage a fundamental change in attitude to quality within a software development organisation.

However, producing yet another standard is not enough if it ends up sitting unused on a shelf. In order to be effective, it must be paired with an enforcement mechanism.

1. *Fagan, 1976; Shull et al, 2002*

IfSQ therefore proposes the adoption of a lightweight quality process using code inspection, consisting of a three-level set of coding standards combined with an assessment method. Together these provide a reliable measure of the quality of a piece of source code. This booklet describes the first of the three levels, IfSQ Level-1, and its associated assessment method.

3. The IfSQ Solution

IfSQ has developed a three-level set of **coding standards**, that can significantly improve the quality practices within any programming team. Each level describes a number of identifiable defects that a programmer should learn to look for and fix. The hierarchy of the levels reflects the relative complexity of identifying and fixing these defects in the software.

To encourage the adoption of these standards, IfSQ has also formulated an **assessment process**, which can be used for self-assessment, peer-assessment, third-party assessment, or full certification issued by a recognized quality body.

3.1. Defects and Defect Indicators

Defects

We define a defect as an error or omission in the source code of a computer program that may cause it to malfunction. There are three ways to find defects in software prior to taking it into use:

- By testing the software
- Through static analysis of the code
- By inspecting the code

In terms of cost, it is cheaper by far to identify and fix defects during the coding phase than finding them during testing or production and having to send the code back through another cycle of bug fixing and testing. If bugs are uncovered by the end-user, the result can be business disruption for the user, with potential financial or legal consequences.

Defect Indicators

It is not always possible to identify defects directly, but there are usually patterns indicating that defects are present. We refer to these patterns as defect indicators.

Through analysis of research papers and from its extensive experience in performing code inspections, peer reviews, and walkthroughs, IfSQ has identified a wide range of defect indicators and encapsulated these into a set of standards that can be applied to any piece of code, from a subroutine to an entire system, written in any language.

Assessing Defect Indicators

When assessing a program, we mark all lines of code affected by each defect indicator. We refer to these as defect lines. The number of defect lines in a program per thousand lines of code is an effective measurement of quality. This measurement can be used in two ways:

- to estimate repair costs
- to set standards for maintenance

3.2. The IfSQ Standards

IfSQ has produced a set of standards for assessing computer program source code. These standards are divided into three levels according to the expertise required and the time it takes to perform an assessment:

- IfSQ Level-1: Entry-Level
- IfSQ Level-2: Foundation-Level
- IfSQ Level-3: Industry Best Practice

The IfSQ Level-1 Standard

The base level for assessing quality is the IfSQ Level-1 Standard for Computer Program Source Code. Level-1 defines the most obvious and commonly occurring defect indicators that are universally acknowledged by software experts as bad practice.

IfSQ Level-1

This standard helps programmers, managers or auditors to locate indicators of low reliability and high maintenance costs in computer software and fix them in an early stage before they proceed to testing or production. Software that meets the requirements of IfSQ Level-1 is more reliable, and easier to maintain, than software that does not meet the requirements.

The Level-1 Standard is described in detail in Section 4.

The IfSQ Level-2 Standard

The IfSQ Level-2 Standard for Computer Program Source Code adds an additional set of defect indicators to those contained in Level-1 to create a more comprehensive measure of software quality.

Aside from the additional defect indicators, there are two main differences between Level-1 and Level-2. Firstly, the two standards require differing levels of expertise in order to identify defects. The defect indicators in Level-1 are sufficiently obvious and easy to identify that anyone, regardless of programming experience, is able to perform a Level-1 assessment. The Level-2 standard on the other hand requires a more advanced level of programming knowledge and experience.

Secondly, whereas Level-1 is essentially objective in its assessment, in Level-2 we actively promote discussion between the assessor and the programmer, by requiring the assessor to use his professional judgement to look for and identify potential issues, which we refer to as Causes for Concern (CfCs), and to resolve his concerns directly with the programmer.

The Level-2 Standard is still concise enough, however, that an assessment can be performed quickly by an individual. Level-2 inspections are extremely cost-effective because the savings made by finding defects at an early stage greatly outweigh the costs of the assessment. The Level-2 Standard is described in more detail in the book *IfSQ Level-2: A Foundation-Level Standard for Computer Program Source Code*.

The IfSQ Level-3 Standard

The IfSQ Level-3 Standard collates an extensive and up-to-date set of defect indicators, including those from Level-1 and Level-2. These indicators are encapsulated in a comprehensive check-list for code walk-throughs, used to perform an in-depth analysis of program source code.

Because the Level-3 Standard represents current industry best practice, it is not set in stone. The checklist is updated every 6 months to reflect any new research. Level-3 is therefore available on a subscription basis.

Compared to the first two levels, IfSQ Level-3 is significantly more expensive and time-consuming to apply, since it requires a substantially higher degree of programming expertise, and also because it requires two people to perform the assessment, due to the extent of the checklist.

For more details, see the book, *IfSQ Level-3: Industry Best Practice for Computer Program Source Code.*

3.3. The IfSQ Assessment Process

While the person most directly impacted by unreliable software is the user, in fact all stakeholders (including programmers, project managers, directors, and shareholders) have an interest in improved quality and reduced risk.

However, without some independent guarantee of quality, there is a higher chance that software may be incomplete, poorly structured, or unquantifiable in terms of risk or future costs of supporting the software.

A process that requires a programmer to sign off on a piece of source code and make a declaration about its perceived quality helps avoid these types of risk by guaranteeing that basic quality processes are being applied in a systematic way.

In addition to the three levels of Standard, IfSQ has therefore defined an assessment process that specifies:

- an inspection method
- an assurance of compliance to the Standards

The following figure illustrates the IfSQ Assessment Process.

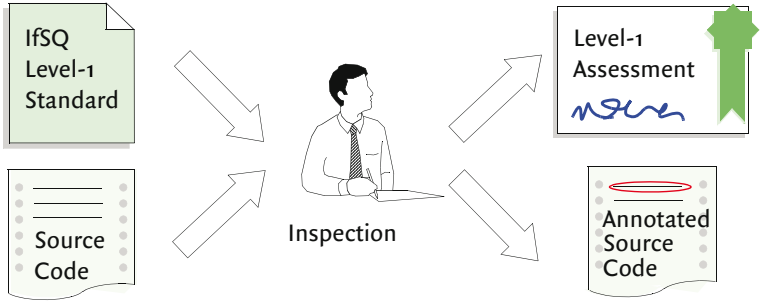


Figure 2. The IfSQ Assessment Process

There are four levels of assurance of compliance:

- **Self-assessment**, an inspection carried out by the programmer himself,
- **Peer-assessment**, an inspection carried out by a member of the development team on a colleague's code,
- **Third-party assessment**, an inspection carried out by a third party uninvolved in the development of the code,
- **Certification**, an inspection by a quality assurance body.

The end product of an IfSQ Assessment or Certification is a completed IfSQ Compliance Assessment Form, as shown on the following page. The inspection method and levels of quality assurance are described in detail in Section 5.

I, THE UNDERSIGNED, HEREBY DECLARE THAT:

1

I have familiarized myself with the Institute for Software Quality Level-1 Standard for Computer Program Source Code *(hereafter "the Standard")*, and the IfSQ Source Code Assessment Method *(hereafter "the Method")*.

2

> Fill in the name of the software

I have assessed the attached software, identified as:

_____

(hereafter "the Software") line by line, in accordance with the Method, and recorded the defect-line count on each page.

3

> Fill in the count for each defect indicator; write the word "zero" if none was found

In my assessment I searched for the defect indicators as defined in the Standard, and have recorded the total defect-line count below:

WORK IN PROGRESS		COUNT
WIP-1	Vague To Do Comment indicating that code needs to be changed, but with no specification as to when	
WIP-2	Disabled Code Code which has been made unreachable (e.g. made into a remark) without explanation	
WIP-3	Empty Statement Block A placeholder for program logic without an explanation as to why it is empty	

STRUCTURED PROGRAMMING		COUNT
SP-1	Routine Too Long A routine (e.g., method, subroutine, or function) consisting of more than 200 lines	
SP-2	Nesting Too Deep Nesting of conditional statements to more than 4 levels	

SINGLE POINT OF MAINTENANCE		COUNT
SPM-1	Magic Numbers Numeric literals, other than zero or one, hard-coded into program logic	

4

> Tick the first box if no defect indicators were found, otherwise, tick the second box

MY ASSESSMENT IS THAT

☐ The Software is free of the defect indicators defined in the Standard, and is therefore IfSQ Level-1 Compliant

☐ The Software is NOT free of the defect indicators defined in the Standard, and is therefore NOT IfSQ Level-1 Compliant

Name: _____ Date: _____

Organisation: _____ Signed: _____

IfSQ

Level-1 Compliance Assessment

www.ifsq.org

Figure 3. IfSQ Level-1 Compliance Assessment Form

4. The IfSQ Level-1 Standard

Categories of Level-1 Defect Indicator

IfSQ has identified a core set of principles that can be applied to the software development process:

- **Complete your work:** When you deliver a program, it should be free of unfinished work.
- **Divide and conquer:** Break down complex programs into smaller programs that are simple enough to understand and maintain.
- **Don't repeat yourself:** Avoid duplicating identical elements in multiple places within the same program.

IfSQ Level-1 formalizes these principles into the following inspection categories:

- **Work In Progress (WIP):** There are clear indications that the program is not yet finished.
- **Structured Programming (SP):** There are clear indications that part of the program is too complex.
- **Single Point of Maintenance (SPM):** Values have been hard-coded into the program.

Each Level-1 defect indicator falls into one of the above categories. This section describes the Level-1 indicators, the risks involved if the indicators are ignored, and a number of solutions which can be applied if the indicators are present.

4.1. Work In Progress (WIP)

Work In Progress means there are indications in the code that the programmer had intended (or is intending) to perform some work, but that this work has not been completed.
At the very least, a work in progress indicator causes confusion for maintenance programmers, wasting their time. At worst, it may indicate missing functionality, which could later lead to software failure.

There are 3 forms of Work in Progress that are easy to detect:

- **Vague "To Do" (WIP-1):** A programmer has left a note to himself or his colleague indicating that a piece of work needs to be done. However it is clear that the work has not been carried out, and there is no indication as to when the work needs to be done.
- **Disabled Code (WIP-2):** Code has been written and the programmer has disabled it, or switched it off, without making it clear why it has been disabled, or when or whether it will be reenabled.
- **Empty Statement Block (WIP-3):** The programmer has left a statement block or placeholder empty. When a programmer designs a program top-down he will often first outline the structure of the program in the form of statement blocks and fill in the content of each block in the course of his work. An empty statement block therefore indicates that there may be missing logic and that some extra code may be required.

WIP-1: Vague “To Do”

> DEFECT INDICATOR

There is a comment indicating that the programmer intends to add a piece of code, but has not specified an exact timeframe or reason, or other precise explanation.

For example, text such as the following are all indications that a program may be incomplete:

- “To do”
- “Not Yet Implemented”
- “Action point”

> RISKS

A “To Do” may indicate missing functionality. In other words, the programmer has at some point decided that code needs to be written, but has not finished the work.

- If there is missing functionality, the problem may be found during testing and need to be fixed, or it may be found after the program goes into production, with unforeseen consequences, such as a crash or malfunction.
- If no code is actually required, a maintenance programmer may later waste time trying to determine whether it is required.

> ASSESSMENT

- Mark all of the lines of the comment block that contains the defect indicator.

> REMEDY

- Add a comment explaining when the work needs to be done, and why, OR
- Do the work, OR
- Determine the work doesn’t need to be done and remove the comment.

Businessactivity.cs

```
2377  private void HandleWorkflowEvents(int eventStatus)
2378  {
2379    foreach (ItemOfBusiness iob in this.Items)
2380    {
2381      foreach (DecisionPoint dp in iob.DecisionPoints)
2382      {
2383  //    if (dp.GetCase() != null)
2384  //    {
2385          try
2386          {
2387            this.EventWebService.HandleEvent(
2388              "StatusUpdateInActivityOccurred",
2389              dp.GetCase().ObjId.ToString() +
2390                ObjId.ToString(),
2391              eventStatus);
2392          }
2393          catch
2394          {
2395            // CAN BE REMOVED FOR PRODUCTION
2396            // Temporarily use deprecated event
2397            try
2398            {
2399              this.EventWebService.HandleEvent(
2400                "StatusUpdateInActivityOccurred",
2401                dp.case.id, eventStatus);
2402            }
2403            catch { }
2404          }
2405  //    }
2406      }
2407    }
2408  }
```

WIP-1

Figure 4. WIP-1 Vague To Do

WIP-2: Disabled Code

> DEFECT INDICATOR

Code has been made unreachable, for example by:

- turning it into a comment or remark,
- placing a return statement above the code, causing the routine to exit without executing the code,

and there is no comment explaining why.

> RISKS

- If the code should not have been disabled and is required in production, the problem may be found during testing and need to be fixed, or it may be found after the program goes into production, with unforeseen consequences, such as a crash or malfunction.
- If the code is commented out because the programmer thinks it may be needed in future, another programmer may later remove it because he is not aware of the first programmer's intentions.
- If the code is actually not required, a maintenance programmer may waste time later figuring this out.

> ASSESSMENT

- Mark all of the disabled lines of code.

> REMEDY

- Determine if the code is required now, later, or not at all:
  - If now, enable it.
  - If later, write a comment to record your decision and the date that you made it.
- Otherwise remove the "dead" code.

Businessactivity.cs

```
2377  private void HandleWorkflowEvents(int eventStatus)
2378  {
2379    foreach (ItemOfBusiness iob in this.Items)
2380    {
2381      foreach (DecisionPoint dp in iob.DecisionPoints)
2382      {
2383  //    if (dp.GetCase() != null)
2384  //    {
2385          try
2386          {
2387            this.EventWebService.HandleEvent(
2388              "StatusUpdateInActivityOccurred",
2389              dp.GetCase().ObjId.ToString() +
2390                ObjId.ToString(),
2391              eventStatus);
2392          }
2393          catch
2394          {
2395          // CAN BE REMOVED FOR PRODUCTION
2396          // Temporarily use deprecated event
2397            try
2398            {
2399              this.EventWebService.HandleEvent(
2400                "StatusUpdateInActivityOccurred",
2401                dp.case.id, eventStatus);
2402            }
2403            catch { }
2404          }
2405  //    }
2406      }
2407    }
2408  }
```

WIP-2

WIP-1

WIP-2

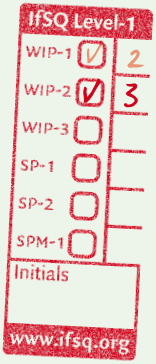


Figure 5. WIP-2 Disabled Code

WIP-3: Empty Statement Block

> DEFECT INDICATOR

There is a placeholder for program logic containing no code and no explanation as to why it is empty. For example, any of the following words or symbols with nothing in between:

- "BEGIN END"
- "IF ENDIF"
- "ELSE ENDIF"
- {}

Or:

- a paragraph containing a return
- a routine with just RETURN FALSE

> RISKS

The empty block may indicate missing functionality. In other words, the programmer has decided at some point that code needs to be written, but has not started the work.

- If there is missing functionality, the problem may be found during testing and need to be fixed, or it may be found after the program goes into production, with unforeseen consequences, such as a crash or malfunction.
- If no code is actually required, a maintenance programmer may waste time later figuring this out.

> ASSESSMENT

- Mark the lines of code that delimit the empty statement block.

> REMEDY

- Find out what code should be there and put it in, OR
- Add a comment to explain why no code is required, OR
- Remove the empty block.

Businessactivity.cs

```
2377  private void HandleWorkflowEvents(int eventStatus)
2378  {
2379    foreach (ItemOfBusiness iob in this.Items)
2380    {
2381      foreach (DecisionPoint dp in iob.DecisionPoints)
2382      {
2383  //    if (dp.GetCase() != null)
2384  //    {
2385          try
2386          {
2387            this.EventWebService.HandleEvent(
2388              "StatusUpdateInActivityOccurred",
2389              dp.GetCase().ObjId.ToString() +
2390                ObjId.ToString(),
2391              eventStatus);
2392          }
2393          catch
2394          {
2395            // CAN BE REMOVED FOR PRODUCTION
2396            // Temporarily use deprecated event
2397            try
2398            {
2399              this.EventWebService.HandleEvent(
2400                "StatusUpdateInActivityOccurred",
2401                dp.case.id, eventStatus);
2402            }
2403            catch { }
2404          }
2405  //    }
2406      }
2407    }
2408  }
```

WIP-2

WIP-1

WIP-3

WIP-2

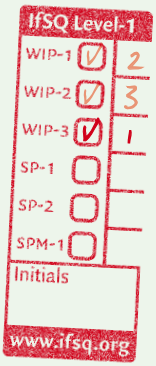


Figure 6. WIP-3 Empty Statement Block

IfSQ Level-1

4.2. Structured Programming (SP)

The costs of producing and maintaining a computer program are largely determined by its complexity.

In writing computer programs a programmer typically uses structured programming techniques to break complex problems down into simpler problems that are easier to understand and solve. This "divide and conquer" process can be repeated until each component is small enough and simple enough to understand, build and maintain.

IfSQ Level-1 contains two easy-to-measure indications that the divide and conquer process has not yet been completed:

- **Too Long (SP-1):** A routine is longer than 200 lines (including comments and blank lines).
- **Too Deep (SP-2):** Nesting of conditional statements is deeper than 4 levels.

SP-1: Routine Too Long

> DEFECT INDICATOR

A program (method, module, routine, subroutine, procedure, or any named block of code) consists of more than 200 lines (including comments and blank lines).

> RISKS

Programs over 200 lines are more error-prone and therefore more expensive to maintain.[2]

> ASSESSMENT

- Mark all the lines following the 200th line of the program, including comments and blank lines.

> REMEDY

- Restructure or refactor your code into smaller, easy-to-understand chunks, OR
- Write a comment justifying the length of the program.

Do NOT:

- Remove comments or blank lines to make it shorter,
- Put multiple commands on one line to make the program shorter.

2. *Basil & Perricone 1984.*

PartyDialog.cs

```
1753            private void ShowAddenda()
1754            {
1755              _showAddendumPage = false;
1756
1757      if(AR83() || AR100())
1758        {
1759        if (_party.Addendum.IsNull)
1760        {
1761          Addendum addendum = Addendum.NewAddendum();
1762          _party.Addendum = addendum;
1763        }
1764
```

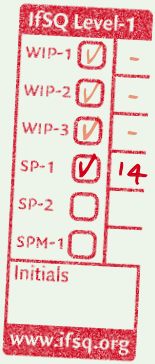
page 39 of 53

186 lines omitted here (pages 40 – 42)

PartyDialog.cs

```
1950              if (_case.Volatility.HasNewAddendum)
1951              {
1952                if ( _party != null &&
1953                     _support == null)
1954                {
1955                  if (_party.IsInvolved)
1956                  {
1957                    if (_party.Addendum.IsNull)
1958                    {
1959                      Addendum addendum =
1960                        Addendum.NewAddendum();
1961                      _party.Addendum = Addendum;
1962                    }
1963                    _addendumControl.Addendum =
1964                      _party.Addendum;
1965                    _showAddendumPage = true;
1966                  }
1967                  else
```

SP-1



page 43 of 53

Figure 7. SP-1: Routine Too Long

SP-2: Nesting Too Deep

> DEFECT INDICATOR

Statements involving a condition have been nested to a depth of more than 4.

> RISKS

With more than 4 levels of nesting, programs become difficult to understand and therefore difficult to maintain. Programmers are more likely to introduce new errors when they make changes.[3]

> ASSESSMENT

- Mark all the lines controlled by the 5th level of condition.

> REMEDY

- Refactor the deeply nested code into its own routine, OR
- Redesign the tests in the condition, OR
- Write a comment justifying the level of nesting.

3. *Yourdon 1986; Ledgard & Tauer 1987.*

PartyDialog.cs

```
1753            private void ShowAddenda()
1754            {
1755              _showAddendumPage = false;
1756
1757      if(AR83() || AR100())
1758        {
1759        if (_party.Addendum.IsNull)
1760        {
1761          Addendum addendum = Addendum.NewAddendum();
1762          _party.Addendum = addendum;
1763        }
1764
```

page 39 of 53

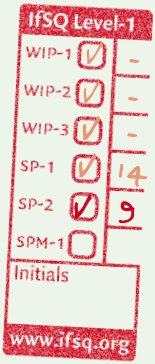
186 lines omitted here (pages 40 – 42)

PartyDialog.cs

```
1950              if (_case.Volatility.HasNewAddendum)
1951              {
1952                if ( _party != null &&
1953                     _support == null)
1954                {
1955                  if (_party.IsInvolved)
1956                  {
1957                    if (_party.Addendum.IsNull)
1958                    {
1959                      Addendum addendum =
1960                        Addendum.NewAddendum();
1961                      _party.Addendum = Addendum;
1962                    }
1963                    _addendumControl.Addendum =
1964                      _party.Addendum;
1965                    _showAddendumPage = true;
1966                  }
1967                  else
```

SP-1

SP-2



page 43 of 53

Figure 8. SP-2: Too Deep

IfSQ Level-1

4.3. Single Point of Maintenance (SPM)

The task of a maintenance programmer is to implement a requested change to a piece of software in a consistent fashion, for example, a change to the calculation of sales tax affecting multiple programs.

This task is made unnecessarily difficult if the program is constructed in such a way that algorithms and values are duplicated throughout the code, for example through the use of copy and paste, or by hard-coding values into a program.

The concept of a single point of maintenance dictates that frequently used elements should be defined, and modified, in a single location. Duplication of such elements increases the difficulty of change, may decrease clarity and increases the likelihood of inconsistency.

Level-1 deals only with hard-coded numeric values:

- **Magic Numbers (SPM-1):** Numeric literals (other than 0 or 1) have been hard-coded into the program.

SPM-1: Magic Numbers

> DEFECT INDICATOR

Numeric literals (other than 0 or 1) have been embedded directly and without explanation into the source code. For example "34" or "86400".

> RISKS

If numbers are embedded in code in this way, it increases the time needed to make maintenance changes, and increases the risk of error.[4]

> ASSESSMENT

- Mark all the lines that contain an unexplained numeric literal other than 0 (zero) or 1 (one).

> REMEDY

Isolate a single copy of the number and refer to it. For example, isolate the number into:

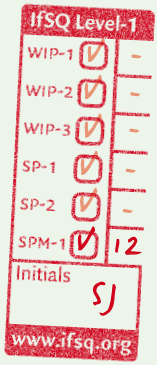
- a local constant or enumerated type
- a global constant or enumerated type
- a constant class
- an initialisation file

Note: If your programming language does not support constants, simulate this, for example, by declaring a variable and initialising it at the beginning of the program.

4. *Korson & Vaishnavi 1986*

printlib.4gl

```
0474  procedure print_header (
0475    input = varchar(132) /* L_INPUT_LINE */ not null ) =
0476
0477  declare
0478    name = varchar(40) /* L_DEPT_NAME */ not null;
0479    owner = integer not null;
0480
0481  begin
0482
0483    --  Print the name..
0484    if left(input, 3) = 'pro' then
0485      name = shift(input, -3);
0486    elseif left(input, 3) = 'acc' then
0487      name = shift(input, -3);
0488    endif;
0489    printfill ( name, 34 );
0490
0491    --  Print the owner..
0492    if left(input, 3) = 'pro' then
0493      owner = 3;
0494    elseif left(input, 3) = 'acc' then
0495      owner = 2;
0496    endif;
0497    if owner <> 0 then
0498      printfill ( decodeowner ( owner ), 34 );
0499    else
0500      printfill ( '', 34 );
0501    endif;
0502
0503    -- Print the rest of the input
0504    if length(input) > 37 then
0505      print( shift(input, -37) );
```



page 12 of 33

Figure 9. SPM-1: Magic Numbers

5. The IfSQ Compliance Assessment Method

Assessment Process

The assessor checks the source code page by page, then makes a written declaration that he or she has checked the code against the Level-1 standard and either found it to be free of defects, or marked each of the defect lines found. The end-result of an assessment is therefore a completed IfSQ Compliance Assessment Form—a declaration of compliance or non-compliance to the standard—and the annotated source code.

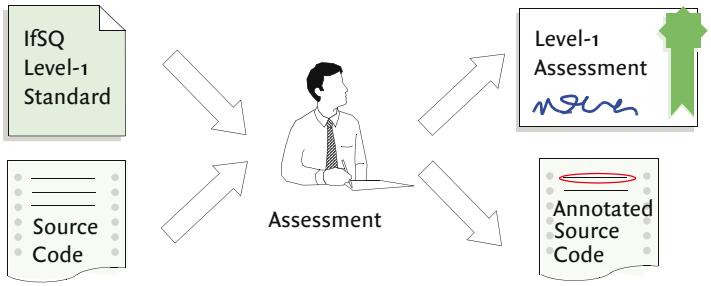


Figure 10. Assessment Process

Who should carry out an assessment?

The IfSQ assessment method can provide various levels of assurance of compliance depending on who is chosen to perform the assessment. Often an assessment is carried out by someone within the development team, such as the programmer himself (self-assessment) or a direct colleague (peer-assessment). For higher levels of assurance, an external organisation or individual can perform the assessment (third-party assessment or certification).

Self-assessment

A self-assessment is carried out by the programmer who wrote the code. The programmer is thereby required to vouch for the quality of his own work by filling in the Compliance Assessment Form.

Peer-assessment

This is a check carried out within the development team by someone other than the person who wrote the code, such as a development team manager or another programmer.

A peer-assessment gives the programmer immediate feedback on any problems that may lie in his code, giving him a chance to fix them before it goes into testing.

Peer-assessment is the most cost-effective approach to quality for the following reasons:

- Knowing that his code will be inspected by others increases a programmer's level of attention to detail and care during programming.[5]
- Having an independent assessor review the code can also uncover unrelated design or construction quality issues.
- The development organisation can commission its own reviews and does not need to wait for, or rely on, external audits to assess quality or risk.

Third-party assessment

A development manager may choose to have an assessment performed by an external organisation or an independent assessor such as an auditor. This has the following benefits:

- It avoids impacting the development process due to the use of scarce programmer resources,
- The assessor can audit separately from the development team, and thus avoid any possible conflict of interest.

5. *Glass, 1999*

Certification

For the highest level of assurance, one can choose to have the assessment carried out by a quality assurance body, who will inspect the software against the Level-1 standard and issue their own certificate of compliance.

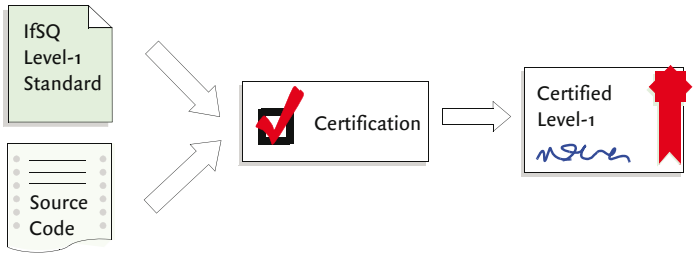


Figure 11. Certification

Ratification

One can choose to have any of the above assessments ratified by an independent person or organisation, such as an external consultant or auditor.

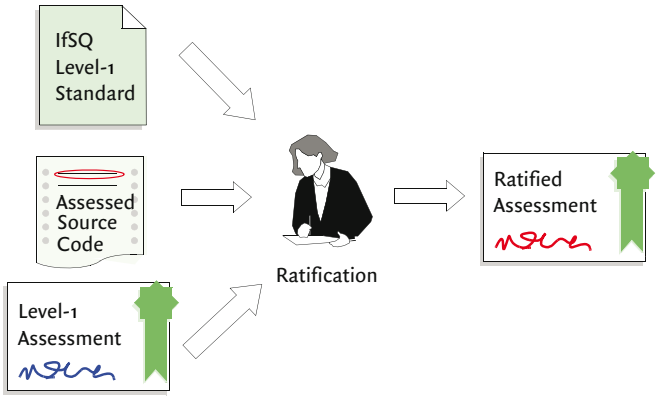


Figure 12. Ratification of an assessment

This ratification consists of a recheck of all or part of the assessed code, to validate the quality of the initial assessment. The ratifier then signs off on the original IfSQ Compliance Assessment Form. For more information on ratification, refer to the IfSQ website.

Which code should you assess, and how much?

There are a variety of reasons you might want to carry out an assessment of a piece of code.

- You might want to assess software that is causing problems, in order to attempt to quantify if the problems stem from a basic lack of quality.
- You might choose to assess software that has a trust issue, for example software that has been outsourced to a third party.

You also have to decide the size of the sample you want to assess. For example, you could assess a representative portion of the code, only the critical routines, routines which been altered recently, routines which have been altered more frequently than others, or even all of the code.

When should you perform an assessment, and how often?

If you are a programmer, you should perform an assessment as soon as you have finished a piece of code, and are satisfied it is complete and functionally correct.

Code should always be assessed before it is submitted to the formal testing procedure, or taken into use.

If you are a programmer performing maintenance, it is also useful to carry out an assessment before you make any changes, in order to assess the quality, and, if necessary, to adjust your estimates as to how long it will take to perform the work.

5.1. Commissioning a Level-1 Assessment

How to commission an assessment

1. Identify the code (i.e., source code file, class, procedure, method, module, etc.) you wish to have assessed.
2. Choose a qualified assessor. This can be:
   - The original programmer
   - Another programmer on the same team
   - A third party

   Ask the assessor to confirm he or she has read and understood the IfSQ Level-1 Standard.
3. Notify the assessor that his assessment may be subject to a review and ratification at a later date and that he should therefore annotate each page exactly as instructed in this document.
4. Once the assessment is complete, check that the IfSQ Compliance Assessment Form has been filled in correctly.

5.2. Performing a Level-1 Assessment

Performing an assessment consists of:

- Inspecting and annotating the source code page by page,
- Filling in the Compliance Assessment Form.

Note: While inspecting each page, you may find it useful to keep a running total of the number of affected lines for each type of defect, for filling in the Compliance Assessment Form later.[6]

How to inspect the source code

1. Print the code (i.e., class, procedure, method, module, etc.) to be reviewed.[7]
2. Take the first page of code, and stamp it using the IfSQ Level-1 stamp.

```
for (int ii=0; ii<100; ii++)
{
  string res = gc.getText("TextTypeLabel"+ii.ToString());
  if (res == null) break;
  ar.Add(res);
}
...
string name = config.StaticGraphImageName;
int pos = name.IndexOf(".gif");
clickurl += "0";
if ( pos >= 0 )
```

6. You can download a pre-printed worksheet from www.ifsq.org.

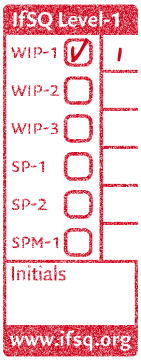
7. Each page will need to be stamped, so if you wish, you can choose to use paper pre-printed with the IfSQ stamp. The pre-printed paper can be downloaded from www.ifsq.org.

3. Assess the page using the criteria for each type of defect indicator as described in Section 4, "The IfSQ Level-1 Standard", as follows:

   a. Scan for any occurrences of WIP-1. If you find one, circle it on the page, and write the defect indicator's abbreviation next to it.

```
sHtml = SHtml & "</tr>" & vbCrLf
if Len(vResult) <> 0 Then
  ReDim Preserve vExport(iIndex)
  vExport(iIndex) = Trim(Replace(vResult, ".", ","))
  Inc iIndex
Else
  'I still need to think about this   WIP-1
End If
```

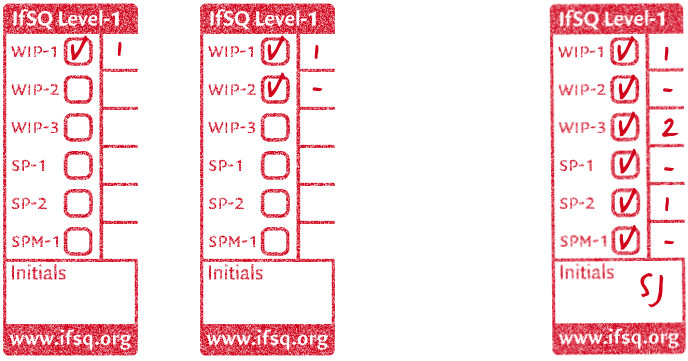
   b. After you have finished scanning the page, tick the associated box on the stamp, and write down the number of defect lines, i.e., the number of lines in which the defect appears. If you found none, enter a dash mark, or a zero.



Pass 1

   c. Repeat steps a-b for each of the remaining five types of Level-1 defect indicator in turn: WIP-2, WIP-3, SP-1, SP-2 and SPM-1.

d. Once you have assessed the page for all six types, initial the stamp.



Pass 1 **Pass 2** **>>>** **Pass 6**

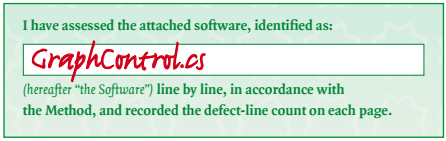
e. Update the running totals of the number of defect lines for each of the six indicators.

4. Repeat Steps 2 and 3 for all other pages. Once you are familiar with the method, this should take approximately 1 minute per page (which equates to about 2 Kb per minute).

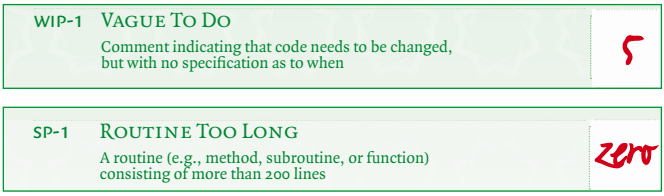
5. On completion of the review, fill out the Compliance Assessment Form, as described below.

How to fill in the IfSQ Level-1 Compliance Assessment Form

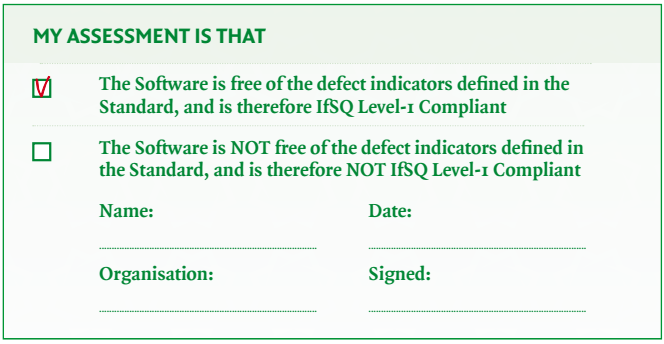
1. Enter the name of the piece of code you have just assessed:



2. Enter the total number of defect lines for each defect type: If there are no defect lines found for a particular category, enter the word "zero".



3. If the totals all say zero, then the software is IfSQ Level-1 compliant. If you have found any defect lines then the software is not Level-1 compliant. Mark one of the choices accordingly in the compliance section.



4. Finally sign and date the form, and attach it to the annotated source code.

6. Research

1. IBM found that each hour of inspection prevented about 100 hours of related work (testing and defect correction) *(Holland 1999)*.
2. A study of large programs found that each hour spent on inspections avoided an average of 33 hours of maintenance work and that inspections were up to 20 times more efficient than testing *(Russell 1991)*.
3. NASA's Software Engineering Laboratory found that code-reading detected 3.3 defects per hour of effort: Testing detected about 1.8 errors per hour. Code reading found 20 to 60% more errors over the life of the project than the various kinds of testing did *(Card 1987)*.
4. As much as 90% of development effort comes after initial release *(Pigoski 1997)*.
5. Construction errors detected in system test cost 10 times more to fix than in construction phase. Construction errors detected post-release cost 10-25 times more to fix than in construction phase *(Fagan 1976; Dunn 1984; Boehm & Turner 2004, Shull et al. 2002)*.
6. Debugging and associated rework takes about 50% of the time spent in a typical software development cycle *(Boehm 1987, Haley 1996, Jones 1998, Shull et al. 2002, Wheeler, Brykczynski & Meesen 1996, Wiegers 2002)*.
7. Large programs that use information hiding are a factor 4 easier to modify than programs which don't *(Korson & Vaishnavi 1986)*.
8. Up to 200 lines of code, routine size is inversely correlated to the number errors per line of code *(Basil & Perricone 1984)*.
9. 39% of all errors are caused by internal interface errors / errors in communication between routines *(Basil & Perricone 1984)*.
10. 50% to 80% of plain "if" statements should have had an "else" clause *(Elshoff 1976)*.
11. Few people can understand more than 3 or 4 levels of nested ifs *(Yourdon 1986; Ledgard & Tauer 1987)*.
12. Control-flow complexity has been correlated with low reliability and frequent errors (McCabe 1976, Shen et al. 1985, Ward 1989).

13. Code reading detected about 80% more faults per hour than testing *(Basili & Selby 1987; Ackerman, Buchwald & Lewski 1989)*.
14. Detection of design defects costs 6 times more using testing than by using inspections *(Basili & Selby 1987, Ackerman, Buchwald & Lewski 1989)*.
15. Average cost of finding an error using code inspections is 3.5 staff hours compared to 15-25 hours to find each error through testing *(Basili & Selby 1987, Ackerman, Buchwald & Lewski 1989)*
16. Increased quality assurance is associated with a decreased error rate but does not increase overall development cost *(Card 1987)*.
17. Software defect removal is the most expensive and time-consuming form of work for software *(Jones 2000)*.
18. Raytheon reduced its cost of rework from about 40% of total project cost to 20% though an initiative that focused on inspections *(Haley 1996)*.
19. ICI found that maintaining a portfolio of about 400 programs was only about 10% of the cost of maintaining a similar set of programs that had not been inspected *(Gilb & Graham 1993)*.
20. Individual inspections typically catch about 60% of defects *(Shull et al. 2002)*.
21. The combination of design and code inspections usually removes 70-85% or more of the defects in a product *(Jones 1996)*.
22. Designers and programmers learn to improve their work through participating in inspections and inspections increase productivity by 20% (Fagan 1976, Humphrey 1989, Gilb & Graham 1993, Wiegers 2002).
23. A study of 13 reviews at AT&T found that the importance of the review meeting itself was overrated; 90% of the defects were found in preparation for the review meeting and only about 10% were found during the review itself *(Glass 1999)*.
24. About 85% of errors can be fixed in a few hours *(Weiss 1975, Ostrand & Weyuker 1984, Grady 1992)*.

7. Bibliography

Ackerman, A. Frank, Lynne S. Buchwald, & Frank H. Lewski 1989. "Software Inspections: An Effective Verification Process." IEEE *Software*, May/June 1989, 31-36.

Basili, V.R. and B.T. Perricone. 1984. "Software Errors & Complexity: An Empirical Investigation." *Communications of the ACM* 27, no.1: 42 - 52.

Basili, Victor R., and Richard W. Selby. 1987. "Comparing The Effectiveness of Software Testing Strategies", *IEEE Transactions on Software Engineering* SE10, no. 6: 728-38.

Bentley, Jon. 1982. *Writing Efficient Programs*. Englewood Cliffs, NJ: Prentice Hall.

Boehm, Barry and Richard Turner. 2004. *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison Wesley.

Boehm, Barry W. 1987. "*Improving Software Productivity.*" IEEE Computer, September, 43-57.

Card, David N. 1987. "A Software Technology Evaluation Program." *Information and Software Technology* 29, no. 6: 291-300

Card, David N., Victor E. Church, and William W. Agresti. 1986. "An Empirical Study of Software Design Practices." *IEEE Transactions on Software Engineering* SE-12. no. 2: 264-71.

Dunn, Robert H. 1984. *Software Defect Removal*, New York, NY McGraw Hill

Elshoff, James L. 1976. "An Analysis of Some Commercial PL/1 Programs." *IEEE Transactions on Software Engineering*" SE-2 no. 2: 113-20.

Endres, Albert. 1975. "An Analysis of Errors and Their Causes in Systems Programs." *IEEE Transactions on Software Engineering* SE-1, no. 2 (June): 140-49

Fagan, Michael E. 1976. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal* 15, no. 3: 182-211.

Gilb, Tom and Dorothy Graham. 1993. *Software Inspection*, Wokingham, England: Addison-Wesley.

Glass, Robert L. 1999. "Inspections – Some Surprising Findings," *Communications of the ACM*, April 1999, 17-19.

Gorla, N., A.C. Benander and B.A. Benander. 1990. "Debugging Effort Estimation using Software Metrics". *IEEE Transactions on Software Engineering* SE-16 no. 2: 233-31.

Grady, Robert B., and Tom van Slack. 1994. "Key Lessons in Achieving Widespread Inspection Use." *IEEE Software*, July 1994.

Haley, Thomas J. 1996. "Software Process Improvement at Raytheon." *IEEE Software*, November 1996.

Holland, D. "Document Inspection as an Agent of Change". *Software Quality Professional*, December 1999: 22-33.

Humphrey, Watts S. 1989. *Managing the Software Process*, Reading, MA: Addison-Wesley.

Jones, Capers. 1996. "Software Defect-Removal Efficiency," *IEEE Computer*, April 1996.

Jones, Capers. 1998. *Estimating Software Costs*, Reading, MA: Addison-Wesley

Jones, Capers. 2000. *Software Assessments, Benchmarks, and Best Practices*, Reading MA. Addison-Wesley.

Korson, Timothy D., and Vijay K. Vaishnavi. 1986. "An Empirical Study of Modularity on Program Modifiability." *Empirical Studies of Programmers*. Norwood, NJ: Ablex.

Ledgard, Henry F. and John Tauer. 1987. *Professional Software*, vol. 2, *Programming Practice*. Indianapolis: Hayden Books.

McCabe, Tom "A Complexity Measure" *IEEE Transactions on Software Engineering*, SE2, no. 4: 308-20.

Pigoski, Thomas M. 1997. *Practical Software Maintenance*, New York, NY: John Wiley & Sons.

Russell, Glen W. "Experience with Inspection in Ultralarge-Scale Developments", *IEEE Software*, vol. 8, no. 1 (January 1991), pp. 25-31.

Shen, Vincent Y., et al. 1985. "Identifying Error-Prone Software – An Empirical Study." *IEEE Transactions on Software Engineering*, SE-11, no. 4: 317-24.

Shneiderman, Ben. 1980. "Exploratory Experiments in Programmer Behavior." *International Journal of Computing and Information Science* 5: 123-43.

Shull, et al 2002. "What We Have Learned About Fighting Defects" *Proceedings Metrics* 2002, IEEE 249-258.

Soloway, Elliot, Jeffrey Bonar, and Kate Elrich. 1983. "Cognitive Strategies and Looping Constructs: An Empirical Study."

Ward, William T. 1989. "Software Defect Prevention Using McCabe's Complexity Metric." *Hewlett-Packard Journal*, April 64 - 68.

Wheeler, David, Bill Brykczynski, and Reginald Meeson. 1996. *Software inspection: An Industry Best Practice*. Los Alamitos, CA: IEEE Computer Society Press.

Wiegers, Karl. 2002. *Peer Reviews in Software: A Practical Guide*. Boston, MA: Addison-Wesley.

Wiegers, Karl. 2003. *Software Requirements*, 2d Ed, Redmond, WA: Microsoft Press.

Woodfield, S. N., H. E. Dunsmore, and V. Y. Shen. 1981. "The Effect of Modularization and Comments on Program Comprehension." *Proceedings of the Fifth International Conference on Software Engineering*, March 1981, 215-23.

Yourdon, Edward. 1986. "*Managing the Structured Techniques: Strategies for Software Development* in the 1990s." 3d ed. New York, NY: Yourdon Press.

IfSQ Level-1

Notes

IfSQ Level-1