



IfSQ Level-2

A Foundation-Level Standard for Computer Program Source Code

Second Edition
November 2009

Graham Bolton
Stuart Johnston

```
276 WHERE mwst_id = g_mwst_id
277 AND
278 AND
279 AND
280 ;
281 mwss_rec mwss_cur%rowtype;
282 BEGIN
283 debug( 'scherm_waarde', 'start' );
284 IF ( p_item IS NOT NULL )
285 THEN
286 IF ( INSTR( p_item, g_block ) = 1 )
287 THEN
288 v_item := p_item;
289 ELSE
290 v_item := g_block||'.'||p_item;
291 END IF;
292 ELSEIF ( p_code IS NULL
293 AND p_presentatie IS NULL
294 AND p_volgorde IS NULL
295 )
296 THEN
297 v_item := NVL( name_in( 'system.trigger_item' ), name_in
298 ELSEIF ( p_presentatie IS NOT NULL
299 AND p_volgorde IS NOT NULL
300 )
301 THEN
302 IF ( p_volgorde <= 4 )
303 THEN
304 v_item := itemnaam
305 ( p_presentatie => p_presentatie, p_volgorde => p_volgorde );
306 ELSE
307 v_item := null;
308 END IF;
309 ELSE
310 OPEN mwss_cur;
311 FETCH mwss_cur
312 INTO mwss_rec;
313 IF ( mwss_cur%found )
314 THEN
315 CLOSE mwss_cur;
316 v_return := scherm_waarde
317 ( p_presentatie => mwss_rec.presentatie
318 , p_volgorde => mwss_rec.volgorde
319 , p_code => mwss_rec.code
```

| IfSQ Level-2 | |
|--------------|--------------------|
| B/F | Page |
| WIP-1 | Work In Progress |
| WIP-2 | Structured Program |
| WIP-3 | Structured Program |
| SP-1 | Structured Program |
| SP-2 | Structured Program |
| SP-3 | Structured Program |
| SPM-1 | Structured Program |
| SPM-2 | Structured Program |
| SPM-3 | Structured Program |
| CIC-NC | Causes for Concern |
| CIC-WR | Causes for Concern |
| CIC-HM | Causes for Concern |
| Initials | |
| www.ifsq.org | |

| Current Routine | |
|-----------------|------|
| SP-1 | SP-2 |
| C/F | |



IfSQ Level-2

A Foundation-Level Standard for Computer Program Source Code

Graham Bolton
Stuart Johnston

Copyright © IfSQ 2005–2009
IfSQ, Institute for Software Quality.
All rights reserved.

Contents

| | |
|---|-----------|
| 1. Management Overview | 3 |
| 2. Background | 5 |
| 3. The IfSQ Solution | 8 |
| 3.1. Defects and Defect Indicators | 8 |
| 3.2. The IfSQ Standards | 9 |
| 3.3. The IfSQ Assessment Process | 11 |
| 4. The IfSQ Level-2 Standard | 13 |
| 4.1. Work In Progress (WIP) | 15 |
| 4.2. Structured Programming (SP) | 23 |
| 4.3. Single Point of Maintenance (SPM) | 31 |
| 4.4. Defensive Programming (DP) | 37 |
| 4.5. Causes for Concern (CfCs) | 41 |
| 5. The IfSQ Compliance Assessment Method | 46 |
| 5.1. Commissioning a Level-2 Assessment | 50 |
| 5.2. Performing a Level-2 Assessment | 50 |
| 6. Acknowledgements | 57 |
| 7. Research | 59 |
| 8. Bibliography | 61 |

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.

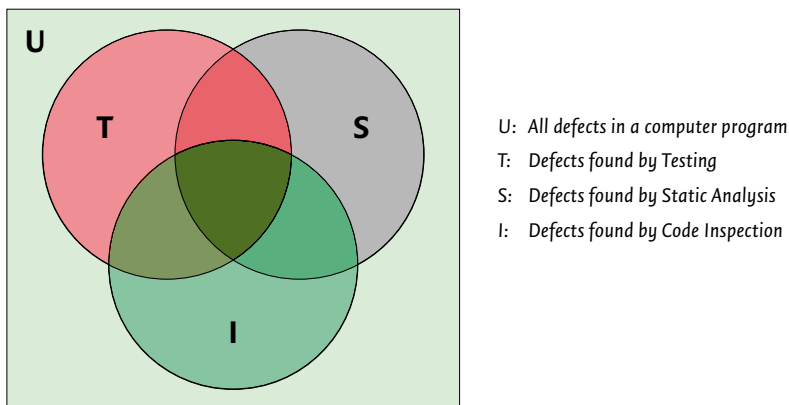


Figure 1. Coverage of software defect detection methods

IfSQ Level-2

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.

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:

- ## Audience

- 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.

```
00001: forIfidStf(eb99e7c17Node:Tablines:Strap...
1: isOTrx()
2: ie in obj10202: f
3: e: Tag: a
4: (codeName: name
5: node: Tag)
6: 0205: span
7: tagAdd()
8: 0206: End: frow new IllegalStateException(
9: 0206: End: frow new IllegalStateException(
```

IfSQ Level-2

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 second of the three levels, IfSQ Level-2, and its associated assessment method.

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.

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.

IfSQ Level-2

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

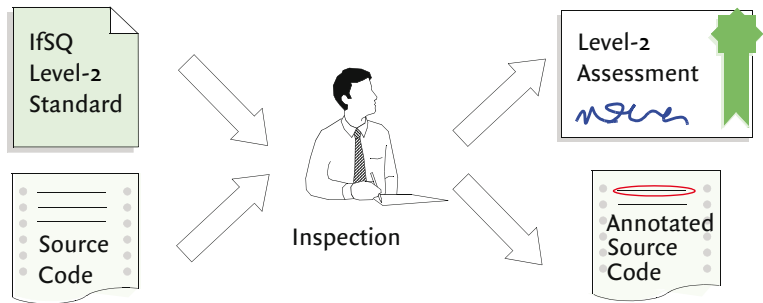


Figure 2. The IfSQ Assessment Process

- **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 inspection method and levels of quality assurance are described in detail in Section 5.

4. The IfSQ Level-2 Standard

Categories of Level-2 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.
- **If it's going to fail, make it fail safely:** Consider all ways the program might fail and take explicit measures to ensure the consequences aren't catastrophic.

IfSQ Level-2 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, or pieces of code have been duplicated at various places.
- **Defensive Programming (DP):** The program does not defend itself against inconsistent data or subsystem failures.

Each Level-2 defect indicator falls into one of the above categories. This section describes the Level-2 indicators, the risks involved if the indicators are ignored, and a number of solutions which can be applied if the indicators are present.

IfSQ Level-2

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.

IfSQ Level-2

■ **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.

> EXAMPLE ASSESSMENT

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 3. WIP-1 Vague To Do

> EXAMPLE ASSESSMENT

```

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) WIP-2
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 WIP-1
2396                     // Temporarily use deprecated event
2397                     try
2398                     {
2399                         this.EventWebService.HandleEvent(
2400                             "StatusUpdateInActivityOccurred",
2401                             dp.case.id, eventStatus);
2402                     }
2403                     catch { }
2404                 }
2405             // } WIP-2
2406         }
2407     }
2408 }

```

Figure 4. WIP-2 Disabled Code

IfSQ Level-2

> DEFECT INDICATOR

- “BEGIN END”
- “IF ENDIF”
- “ELSE ENDIF”
- {}

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

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.

- ## > ASSESSMENT

- ## > REMEDY

- <20>

> EXAMPLE ASSESSMENT

```

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 WIP-1
2396                     // Temporarily use deprecated event
2397                     try
2398                     {
2399                         this.EventWebService.HandleEvent(
2400                             "StatusUpdateInActivityOccurred",
2401                             dp.case.id, eventStatus);
2402                     }
2403                     catch { } WIP-3
2404                 }
2405             // } WIP-2
2406         }
2407     }
2408 }

```

Figure 5. WIP-3 Empty Statement Block

IfSQ Level-2

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-2 contains three simple 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.
- **Routine Too Complex (SP-3):** A routine includes more than 10 comparisons.

IfSQ Level-2

■ 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.

> EXAMPLE ASSESSMENT

| 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 |
| page 43 of 53 | |

SP-1

Figure 6. SP-1: Routine Too Long

IfSQ Level-2

> DEFECT INDICATOR

> RISKS

> ASSESSMENT

- ## > REMEDY

3. Yourdon 1986; Ledgard & Tauer 1987.

> EXAMPLE ASSESSMENT

```

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
page 43 of 53

```

SP-1

SP-2

Figure 7. Nesting Too Deep

```

00001: for If first If (objLevel7Node.LinesString)
00002:   If (objLevel7Node.Nodes.Add(.Label & " " & objLevel7Node.LinesString) = 0)
00003:     objLevel7Node.Nodes.Add(.Label & " " & objLevel7Node.LinesString)
00004:   End If
00005:   objLevel7Node.Nodes.Add(.Label & " " & objLevel7Node.LinesString)
00006: End If

```

IfSQ Level-2

■ SP-3: Logic Too Complex

> DEFECT INDICATOR

A program (method, module, routine, subroutine, procedure, or any-named block of code) contains more than 10 binary terms in conditional statements. This is based on McCabe's Complexity metric.

> RISKS

Programs containing more than 10 comparisons are more difficult to understand and therefore more difficult to maintain. Programmers are more likely to introduce new errors when they make changes.⁴

> ASSESSMENT

- Locate and count all binary terms in the program. An easy way to do this is by looking for the keywords and symbols that can precede them in the language you are using. For example:
 - If, While, Until, etc.
 - And, Or, &&, ||, etc.
- Mark all the binary terms after the 10th occurrence.

> REMEDY

- Simplify complicated tests with Boolean function calls, OR
- Refactor the complex code into separate routines, OR
- Restructure the program to remove any unnecessary repetition.

4. McCabe 1976; Shen et al. 1985; Ward 1989.

> EXAMPLE ASSESSMENT

```

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
page 43 of 53

```

Figure 8. SP-3: Logic Too Complex

IfSQ Level-2

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.

There are three contraventions of Single Point of Maintenance that are easy to detect:

- **Magic Numbers (SPM-1):** Numeric literals (other than 0 or 1) have been hard-coded into the program.
- **Magic Strings (SPM-2):** A string literal has been hard-coded into a statement that influences the flow of a program.
- **Copy/Paste Programming (SPM-3):** An identical or largely similar section of code appears in two or more places in the program or set of programs.

IfSQ Level-2

> DEFECT INDICATOR

> RISKS

> ASSESSMENT

- ## > REMEDY

- 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.

> EXAMPLE ASSESSMENT

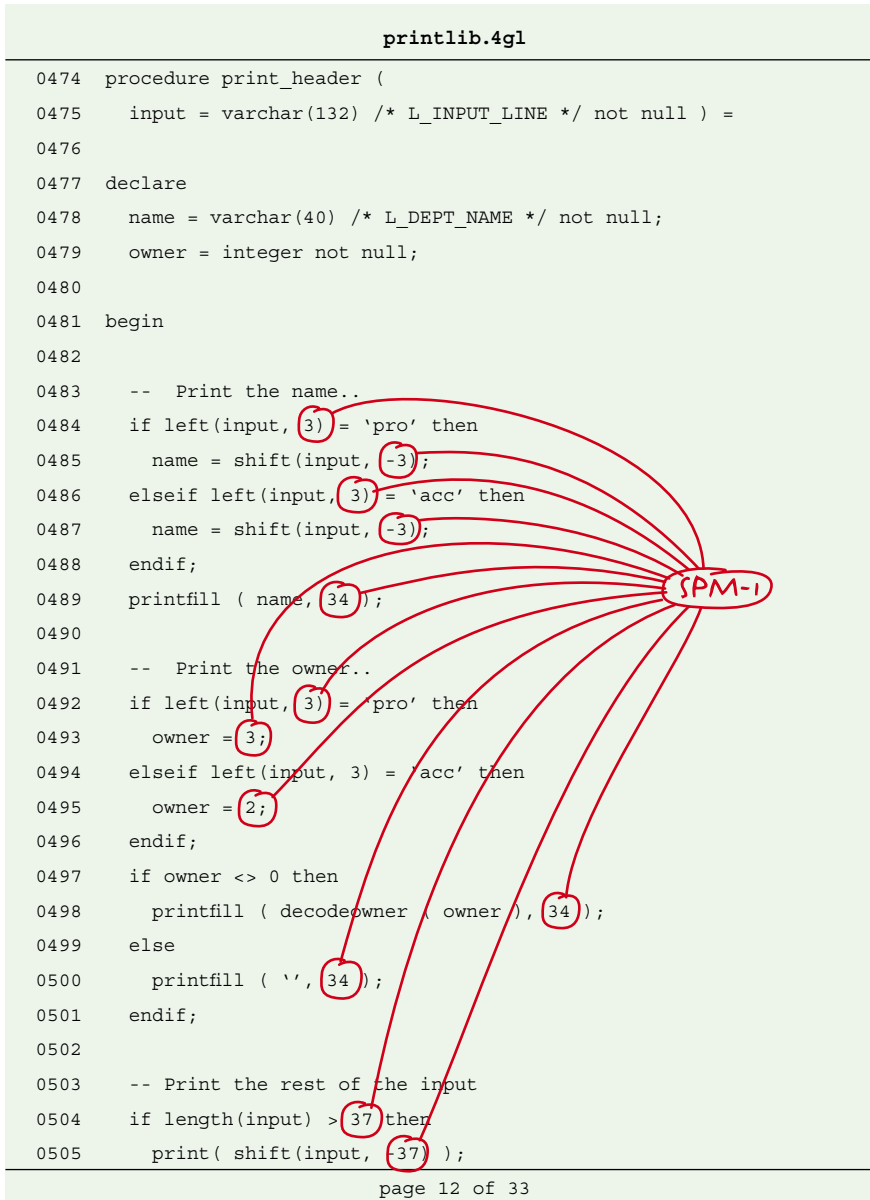


Figure 9. SPM-1: Magic Numbers

IfSQ Level-2

> EXAMPLE ASSESSMENT

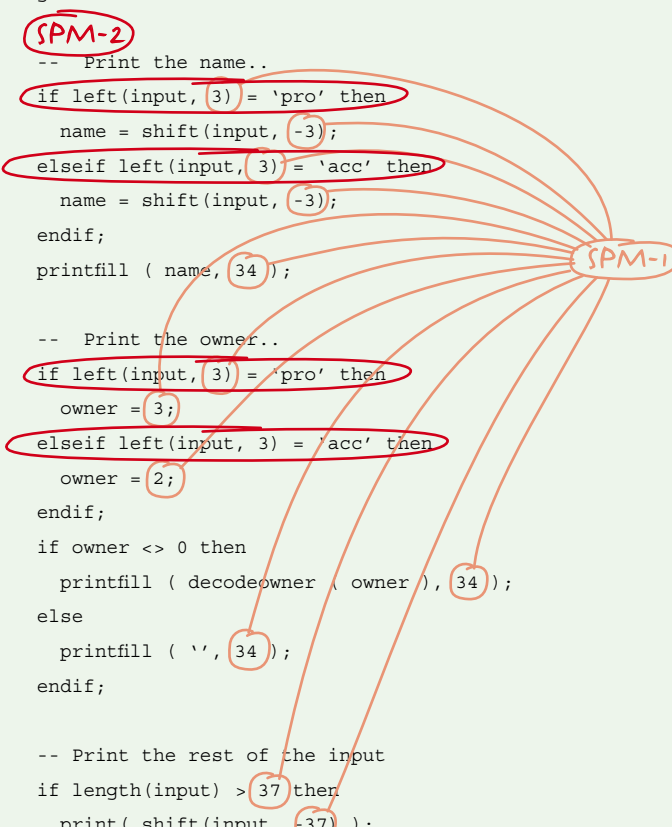
printlib.4gl

```

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     SPM-2
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) );

```

SPM-1



page 12 of 33

Figure 10. SPM-2: Magic Strings

IfSQ Level-2

A largely similar or identical section of code appears in two or more places in the program or set of programs.

Having to modify identical code in multiple places:

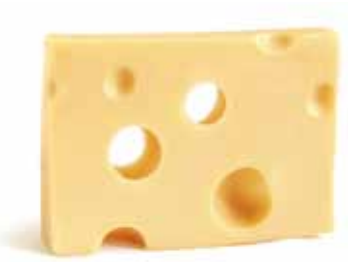
- increases the likelihood of making slightly different modifications under the mistaken assumption that you have made identical ones,
- increases the time needed to make changes,
- increases maintenance costs in direct proportion to the number of times the code has been copied/pasted.

- If you see a block of code that looks familiar, backtrack through the already assessed code looking for similar blocks. If the blocks could be implemented as a (parameterised) subroutine, mark the second and subsequent blocks.

- Isolate a single copy of the code into a separate program (e.g., method, function, subroutine) and reuse it by calling it from the places in which it was used.

4.4. Defensive Programming (DP)

In focussing on the main logic of a program, programmers may fail to take into account abnormal situations, such as invalid data input, or a hard disk becoming full. As a result, their programs are vulnerable to unexpected events or conditions. In essence, such programs have holes in their defenses.



In particular, interfaces between programs are some of the most error-prone areas in a system. One often-cited study⁶ found that 39% of all software errors were internal interface errors, i.e., errors in communication between programs.

To ensure more robust programs, we need not only to protect ourselves from our own mistakes, but also to isolate our programs from the potential errors of others. We refer to this approach as Defensive Programming.

There are three important causes of program malfunction that we can defend against:

- **Parameter Not Checked (DP-1):** A parameter received by a program is used without first checking if its contents are present and within the expected range.
- **Status Ignored After Call (DP-2):** Error status codes or exceptions from the run-time environment are suppressed or ignored, masking internal processing errors.
- **Unexpected State Not Trapped (DP-3):** Part of a program that uses a value to switch between different branches does not trap unexpected cases.

6. Basili and Perricone, 1984.

IfSQ Level-2

■ DP-2: Status Ignored After Call

> DEFECT INDICATOR

An exception is suppressed or the error status returned by a called program is ignored and there is no comment explaining why. Example indicators are:

- **C:** Any call to a file system routine where the result is not checked.
- **SQL:** Any command not immediately followed by a check on the SQL state variable.
- **C++/Java:** An empty statement block following a catch.

> RISKS

- A program may fail silently, in other words, it will continue processing when it should have stopped, with potentially disastrous consequences, such as data corruption or loss.
- Transient information important for tracing the source of the error is lost, making debugging difficult if not impossible.

> ASSESSMENT

- For each of the program and subsystem calls made by program, check to see how the status resulting from the call is used. If it is ignored or suppressed without explanation, mark the call.

> REMEDY

- Put in code to perform error handling immediately following any call to other programs or subsystems.
- In code where exception handling is used, be specific as to which exceptions, if any, should be caught.

IfSQ Level-2

> DEFECT INDICATOR

- A Case or Switch statement does not include a default case.
- A chain of If/Then/Else statements is not terminated by an unqualified Else statement.

> RISKS

- ## > ASSESSMENT

- ## > REMEDY

7. Elshoff 1976

4.5. Causes for Concern (CfCs)

The quickest and most cost-effective way to inspect software is to read it. Studies have shown that reading source code typically catches 60% of defects, is 20% more effective than testing and that each hour spent reading avoids 33 hours of maintenance work.⁸

In the preceding chapters we have described the defect indicators that we look for in Level-2. These are clear-cut, objective, and can be identified in a reasonably mechanical fashion.

In addition to these defect indicators, Level-2 adds an extra level of inspection that requires a deeper understanding of the program and a higher degree of professional skill. The process is decidedly different from looking for defect indicators—you now have to understand the program.

Assessment Aspects

IfSQ has identified three points of view from which a program can be read. These aspects are of crucial importance to the quality of programs and in reducing the Total Cost of Ownership (TCO) of a software system.

- **Completeness:** Is the program functionally complete?
- **Correctness:** Is the program logically correct?
- **Maintainability:** Is the program easy to maintain?

In IfSQ Level-2 we require an assessor to inspect the source code from each of these three points of view in turn. If, in the professional opinion of the assessor, there are any indications that the code violates any of these objectives, this is noted as one of the following Causes for Concern:

8. Russell 1991; Shull et al 2002

■ CfC-1: Not Complete

> PERSPECTIVE

We read the source code from the point of view of completeness, looking for any indication that a program is not complete, which has not yet been flagged with one of the Defect Indicators.

> ASSESSMENT POINTS

The following are examples of things to bear in mind when assessing a program from the aspect of completeness:

- Is each input parameter used?
- Is each output parameter used?
- Does the code check for malicious input (e.g., SQL injection, HTML injection)?
- Are all declared variables being used?
- For each If/Then statement, is the Else clause present?
- In a Case statement in C/Java does the end of each Case have a break?
- Do potentially infinite loops use a safety counter?
- Do recursive routines use a safety counter?

IfSQ Level-2

> PERSPECTIVE

> ASSESSMENT POINTS

- If the routine is a function does it return a valid value in all possible circumstances?
- Does the code initialise variables as they are called if possible?
- Are implicit data type conversions obvious?
- Does the code avoid mixed-type comparisons?
- Do expressions that use integer division work the way they are meant to?
- Do integer divisions avoid integer overflow problems?
- Does the code systemically prevent rounding errors?
- Are all array indices within the bounds of the array?
- Does the code check pointers for validity before using them?

■ CfC-3: Hard to Maintain

> PERSPECTIVE

We read the source code once more, imagining ourselves in the role of a maintenance programmer seeing the program for the first time, looking for any indication that a program may be difficult to maintain, which has not yet been flagged with one of the Defect Indicators.

> ASSESSMENT POINTS

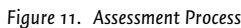
The following are examples of things to bear in mind when assessing a program from the aspect of ease of maintenance:

- Does the routine name describe everything that the routine does?
- Does the routine have strong functional cohesion—doing one and only one thing and doing it well?
- Does this code belong in this application or should it be part of an existing library?
- Do all variables have the smallest scope possible?
- Do names fully and accurately describe the object they represent?
- Are the following names likely to cause problems:
 - names that are misleading
 - names with similar meanings
 - names that are only different by one or two characters
 - names that sound similar

IfSQ Level-2

Assessment Process

The end-result of an assessment is therefore an IfSQ Compliance Assessment Report—a measurement of compliance or non-compliance to the standard—and the annotated source code.



The IfSQ assessment method can provide various levels of assurance of compliance depending on who is chosen to perform the assessment.

<46>

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.¹⁰
- 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.

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 soft-

10. Glass, 1999

IfSQ Level-2

ware against the Level-2 standard and issue their own certificate of compliance.

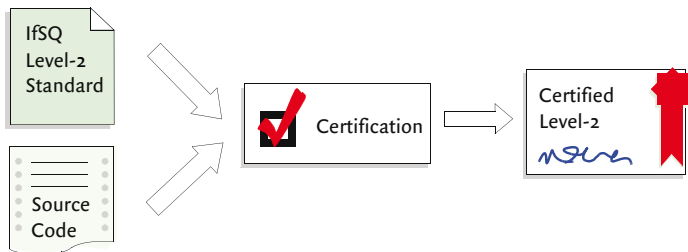


Figure 11. Certification

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.

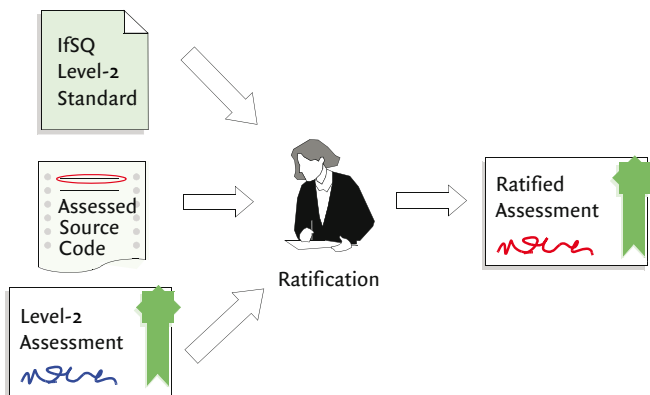


Figure 12. Ratification of an assessment

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 Report. 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.

IfSQ Level-2

How to commission an assessment

- ## 5.2. Performing a Level-2 Assessment

- Printing the source code,
- Inspecting and annotating the source code page by page,
- Producing a Compliance Assessment Report.

The IfSQ Assessment Process requires that you print off the entire piece of code being assessed. Doing the inspection on paper makes it easier to keep a count of defects found and minimises the chance of mistakes. In addition, since the inspected piece of code is an auditable item of work, it must be possible to store it as a physical object.

The IfSQ Assessment Process makes use of a page inspection grid that lets you keep a running total of defects found.

| IfSQ Level-2 | | | |
|-----------------------------|-----|------|-----|
| | B/F | Page | C/F |
| Work In Progress | | | |
| WIP-1 | | | |
| WIP-2 | | | |
| WIP-3 | | | |
| Structured Programming | | | |
| SP-1 | | | |
| SP-2 | | | |
| SP-3 | | | |
| Single Point of Maintenance | | | |
| SPM-1 | | | |
| SPM-2 | | | |
| SPM-3 | | | |
| Defensive Programming | | | |
| DP-1 | | | |
| DP-2 | | | |
| DP-3 | | | |
| Causes for Concern | | | |
| CfC-NC | | | |
| CfC-WR | | | |
| CfC-HtM | | | |
| Initials | | | |
| | | | |
| www.ifsq.org | | | |

Figure 14. IfSQ Level-2 Page Inspection Grid

This inspection grid must be printed on every page of source code (preferably in red so that the code is still visible if there is an overlap). To ensure this you should use paper pre-printed with the grid. On the IfSQ website you can:

- Download a PDF template,
- Order pre-printed paper,
- Download a graphic file of the grid for insertion as a watermark in your word-processing file.

Note: Overlaps can be minimised by printing the code in landscape format.

```
0205         e17Node.Nodes.Count - 1).Tag = .ID
0206     End If
    Throw new InvalidOperationException("Invalid")
End Sub
```

<52>

6. Fill in the C/F (Carried Forward) fields by adding the Page field to the B/F field.

| IfSQ Level-2 | | | |
|------------------|-----|------|-----|
| | B/F | Page | C/F |
| Work In Progress | | | |
| WIP-1 | — | 3 | 3 |
| WIP-2 | | | |
| WIP-3 | — | | |

7. Repeat steps 4 to 6 for the remaining 11 objective inspection criteria (WIP, SP, SPM, DP). (For details of each criteria, see Section 4., “The IfSQ Level-2 Standard”.)
8. Repeat steps 4 to 6 for the 3 subjective inspection criteria. For details see “Assessing Causes for Concern” on page 55.
9. Once you have finished scanning the page, initial the bottom of the grid.

| | | | |
|--------------|--|--|--|
| CfC-WR | | | |
| CfC-HtM | | | |
| Initials | | | |
| GB | | | |
| www.ifsq.org | | | |

10. Finally, copy the C/F values into the B/F fields on the following page.

| IfSQ Level-2 | | | |
|------------------|-----|------|-----|
| | B/F | Page | C/F |
| Work In Progress | | | |
| WIP-1 | — | 3 | 3 |
| WIP-2 | — | 1 | 1 |
| WIP-3 | — | 1 | 1 |

Page 1

| IfSQ Level-2 | | | |
|------------------|-----|------|-----|
| | B/F | Page | C/F |
| Work In Progress | | | |
| WIP-1 | 3 | | |
| WIP-2 | 1 | | |
| WIP-3 | | | |

Page 2

11. Repeat steps 4 to 10 for all remaining pages.

```
0205 Space: 29 (4-Description)
0206 Node.Nodes.Count - 1).Tag = .ID
0206 EndIf row new IllegalStateException("Node
```

In addition to the main inspection grid, there are two mini-grids for SP-1 and SP-3 at the top and bottom of the page.

| | | |
|-----|------|------|
| | SP-1 | SP-3 |
| B/F | | |

| | | |
|-----------------|------|------|
| | SP-1 | SP-3 |
| Current Routine | | |
| C/F | | |

These help you to keep running totals for SP-1 and SP-3

- According to SP-1, a routine may not exceed 200 lines. For every line over this number, the routine receives a mark of 1 on the inspection grid. So if the routine is 220 lines long for example, it receives a score of 20. If a routine continues onto the next page, enter the number of lines in the Current Routine field, then add it to the B/F field to fill in the C/F field.

| | | |
|-----|------|------|
| | SP-1 | SP-3 |
| B/F | 11 | |

| | | |
|-------------------|------|------|
| | SP-1 | SP-3 |
| + Current Routine | 51 | |
| = C/F | 62 | |

Once a routine has reached 200 lines, simply put a cross in the B/F field on subsequent pages and continue adding using the SP-1 field in the main grid.

| | | |
|-----|------|------|
| | SP-1 | SP-3 |
| B/F | X | |

| | | |
|-----------------|------|------|
| | SP-1 | SP-3 |
| Current Routine | X | |
| C/F | X | |

| | Structured Programming | | |
|------|------------------------|----|---|
| SP-1 | | 13 | — |
| SP-2 | | | |
| SP-3 | | | |

- According to SP-3, routines may not contain more than 10 binary terms in conditional statements. For every additional binary term the routine receives a mark of 1 on the inspection grid. If a routine continues on the next page, enter the number of binary terms in the Current Routine field, then add it to the B/F field to fill in the C/F field.

| | | |
|-----|------|------|
| | SP-1 | SP-3 |
| B/F | | 3 |

| | | | |
|---|-----------------|------|------|
| + | Current Routine | SP-1 | SP-3 |
| = | C/F | 8 | 11 |

- Once you have finished scanning the page, copy the C/F values into the B/F fields on the following page.

| | | |
|-----|------|------|
| | SP-1 | SP-3 |
| B/F | 11 | 3 |

| | | | |
|---|-----------------|------|------|
| + | Current Routine | SP-1 | SP-3 |
| = | C/F | 51 | 6 |

| | | |
|-----|------|------|
| | SP-1 | SP-3 |
| B/F | 62 | 9 |

Page 11 Page 12

- Once a routine has reached 10 binary terms, simply put a cross in the B/F field on subsequent pages and continue adding using the SP-3 field in the main grid.

Assessing Causes for Concern

In this part of the inspection, you are looking for defects that are not covered by the IfSQ Level-2 objective defect indicators, but which, in your professional opinion, present a hazard.

Estimate the number of lines that have to be altered to address your concerns and write it in the appropriate field on the inspection grid.

IfSQ Level-2

6. Acknowledgements

The authors wish to acknowledge the following individuals for their contributions and cooperation in the development and production of this standard.

- Steve McConnell, CEO and Chief Software Engineer at Construx Software, for his excellent book “Code Complete” (Microsoft Press, ISBN 0-7356-1967-0) which we used as a starting point for finding research relevant to our subject area.
- Hans Suerink, Chairman of the Technical Policy Committee of NOREA, the Dutch Order of Registered EDP Auditors, who helped us to understand the importance of formal, objective standards in the EDP audit arena.

IfSQ Level-2

7. 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).

IfSQ Level-2

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% through 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).

8. 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/I 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.

IfSQ Level-2

- 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-2

Notes


```
00001: ForIf1stIf (objLevel1Node.Nodes.Add(, Label, WIP-1))
00002: ForIf2ndIf (objLevel1Node.Nodes.Add(, Label, WIP-2))
00003: ForIf3rdIf (objLevel1Node.Nodes.Add(, Label, WIP-3))
00004: ForIf4thIf (objLevel1Node.Nodes.Add(, Label, SP-1))
00005: ForIf5thIf (objLevel1Node.Nodes.Add(, Label, SP-2))
00006: ForIf6thIf (objLevel1Node.Nodes.Add(, Label, SP-3))
00007: ForIf7thIf (objLevel1Node.Nodes.Add(, Label, SPM-1))
00008: ForIf8thIf (objLevel1Node.Nodes.Add(, Label, SPM-2))
00009: ForIf9thIf (objLevel1Node.Nodes.Add(, Label, SPM-3))
00010: ForIf10thIf (objLevel1Node.Nodes.Add(, Label, DP-1))
00011: ForIf11thIf (objLevel1Node.Nodes.Add(, Label, DP-2))
00012: ForIf12thIf (objLevel1Node.Nodes.Add(, Label, DP-3))
00013: ForIf13thIf (objLevel1Node.Nodes.Add(, Label, CFC-NC))
00014: ForIf14thIf (objLevel1Node.Nodes.Add(, Label, CFC-WR))
00015: ForIf15thIf (objLevel1Node.Nodes.Add(, Label, CFC-HtM))
00016: EndIf1stIf
00017: EndIf2ndIf
00018: EndIf3rdIf
00019: EndIf4thIf
00020: EndIf5thIf
00021: EndIf6thIf
00022: EndIf7thIf
00023: EndIf8thIf
00024: EndIf9thIf
00025: EndIf10thIf
00026: EndIf11thIf
00027: EndIf12thIf
00028: EndIf13thIf
00029: EndIf14thIf
00030: EndIf15thIf
00031: EndSub
00032: EndModule
```

Ifsq Level-2

Quick Reference

Work In Progress

- WIP-1—Vague “To Do”
- WIP-2—Disabled Code
- WIP-3—Empty Statement Block

Structured Programming

- SP-1—Routine Too Long
- SP-2—Nesting Too Deep
- SP-3—Routine Too Complex

Single Point of Maintenance

- SPM-1—Magic Numbers
- SPM-2—Magic Strings
- SPM-3—Copy/Paste Programming

Defensive Programming

- DP-1—Parameter Not Checked
- DP-2—Status Ignored After Call
- DP-3—Unexpected State Not Trapped

Causes for Concern

- CFC-NC—Not Complete
- CFC-WR—Wrong Result
- CFC-HtM—Hard to Maintain

| Ifsq Level-2 | | |
|-----------------------------|------|-----|
| B/F | Page | C/F |
| Work In Progress | | |
| WIP-1 | | |
| WIP-2 | | |
| WIP-3 | | |
| Structured Programming | | |
| SP-1 | | |
| SP-2 | | |
| SP-3 | | |
| Single Point of Maintenance | | |
| SPM-1 | | |
| SPM-2 | | |
| SPM-3 | | |
| Defensive Programming | | |
| DP-1 | | |
| DP-2 | | |
| DP-3 | | |
| Causes for Concern | | |
| CFC-NC | | |
| CFC-WR | | |
| CFC-HtM | | |
| Initials | | |
| | | |
| www.ifsq.org | | |



Institute for Software Quality
www.ifsq.org