1. Assert early and assert often

assertion

lines

Assertions today play a vital role in Microsoft program development and test. In a successful product like Office, around one line in ten of the code base is an assertion. In other products, there may be as few as one in a hundred. This wide disparity of ratios suggests that the quality of our products might be improved simply by spread of current good practices from one product team to another.

how to do this. I will present

The purpose of this presentation is to describe a range of assertion macros currently available to users of the program analysis tools PREfix and PREfast. I suggest how you can use them more effectively and more often, from the earliest to the latest stages of product development.

will

2. Engineering test probes

The main use of assertions today is to assist in program test and fault diagnosis. In all branches of engineering, product test is an essential prerequisite before release to manufacture of a new or improved product. For example, in the development of a new aero jet engine, an early working model is installed on an engineering test bench for exhaustive trials. This model engine will first be thoroughly instrumented by insertion of test probes at every accessible internal interface. A rigorous test schedule is designed to exercise the engine at all the extremes of its intended operating range. By continuously checking tolerances at all the crucial internal interfaces, the engineer detects incipient errors immediately, and never needs to test the assembly as a whole to destruction. By continuously striving to improve the set points and tighten the tolerances at each internal interface, the quality of the whole product can be gradually raised. That is the reason for the success of the six sigma quality improvement initiative.

7½ x5 = 37 mins

In the engineering of software, it is the assertions at the interfaces between modules of the program that play the same role as test probes in engine design. My analogy with tightening tolerances suggests that programmers who wish to ~~write~~ deliver trustworthy programs should gradually increase the number and strength of the assertions in code. Paradoxically, the intended effect of assertions is to make a system **more** likely to fail under test; but the reward is that failure is much **less** likely after shipping to the customer.

3. Macros

The defining characteristic of an engineering test probe is that it is removed from the engine before manufacture and delivery to the customer. In computer programs, this effect is achieved by means of conditionally defined macros. The macro is resolved at compile time in one of two ways, depending on a compile-time switch, which is called DEBUG, set for a debugging run, and unset when compiling retail code. My slide shows an example taken from a collection of macros written by Jon Pincus, and made available with the program analysis tools PREfix and PREfast. An assertion may be placed anywhere in the middle of executable code by means of this CONSISTENCY CHECK macro. The details of the whole package are posted on the sharepoint site for this lecture. ?

4. Explanations

The whole point of an assertion is that the programmer should have a good reason for believing that it will always be true when the ~~program is executed~~ reaches that point in its execution. The programmer should be willing to explain the reason why the assertion is valid, in an informally

5 mins

assigned to deal with it 3

written text ~~passed as a second argument to the CHECK macro~~. In case of error, discovered perhaps much later, the programmer will have the information needed to trace the underlying cause of the mistake. This should make correction much easier: but more than that. The programmer will be warned of other likely occurrences of a similar error in the existing code; and will be encouraged to improve the rigour of the its reasoning, to avoid all such errors in the future.

there is a chance that the originator of the error

That is why the PREfix assertion macro requires a second argument ~~parameter~~, a string ~~in~~ which ~~the programmer~~ can explains quite informally the reason why the first parameter will always be true. The more obscure the reason, the greater the value of the explanation. Unexplained assertions can be seen from records in the RAID ~~program defect~~ database to generate grief when they fire in later releases of the same program.

5. Documentation

A major concern of our Company is the continuous evolution and improvement of old code to meet new market needs. Even quite trivial assertions, like that shown on this slide, give added value when the time comes to change the code for the next release. One Development Manager recommends that for every bug corrected in test, an assertion should be added to the code which will fire if that bug ever occurs again. My recommendation is even stronger. From the beginning, there should be enough assertions in the code to ensure that nearly all bugs will be caught by assertion failure – much easier to diagnose than any other kind of failure.

Some developers are willing to spend a whole day to design precautions that will avoid a week's work by a less experienced programmer, tracing an error that may be introduced by a later change to the code. Success in such documentation by assertions depends on long experience and careful judgment in predicting the most likely errors a year or more from now. Not everyone can spare the time to do this under pressure of tight delivery schedules. But it is likely that a liberal sprinkling of assertions in the code will increase the accumulated value of Microsoft legacy, when the time comes to develop a new release

6. Assumptions

In the early testing of a prototype program, the developer wants to check out the main paths in the code before dealing with all the exceptional conditions that may occur in practice. In order to document such a development plan, PREfast provides a variety of assertion which is called a simplifying assumption. The quoted assumption documents exactly the cases which the developer is not yet ready to treat, and it also serves as a reminder of what remains to do later. Violation of such assumptions in test will simply cause a test case to be ignored, and should not be treated as an error. But the priority of the test case should be increased, to ensure that the eventual special case code will be adequately ~~tested~~ exercised. Of course, in a retail build when the debug flag is not set, th~~e~~is macro will give rise to a compile-time error; it will not just be ignored like an ordinary assertion. This gives protection against the risk incurred by more informal TO DO comments, which occasionally and embarrassingly find their way into retail code.

bes

7. Compile-time

All the best debug messages are those given at compile time, since that avoids all the hassle of diagnosis of errors by test. In the Windows product team, a special class of assertion has been implemented called a compile-time check, because it can be evaluated at compile time. The compile time error message is

10 mins

generated by a macro that compiles to an invalid declaration (negative array bound) in C in the case that the compiler evaluates the assertion to false; of course, the assertion must be one that uses only values and functions computable by the compiler. (The compiler will still complain if not.) The example ~~above~~ on this slide shows a test of conformity of the size of two array parameters x and y. Note that this macro is not provided by PREfast.

Of course, only a very few assertions can be evaluated at compile time – at present. To change this is exactly the long-term goal of my colleagues in Microsoft Research, who are developing new programmer productivity tools. By more sophisticate d program analyses, it is increasingly possible to guarantee with mathematical certainty that each assertion will be true on every occasion that it is evaluated. If this guarantee cannot be given, the tool should ideally generate a test case automatically that will expose the fault. Of course, an assertion that has been proved to be always true can be optimised away, to avoid the overhead of evaluation, even on test runs. Because it is known that there will be no errors left for run-time testing. At this Techfest, in the Software Engineering section, Just across the hallway from here you will be able to see the progress the we have made towards this goal.

8. Invariants

Assertions are particularly valuable for documenting object-oriented programs. An invariant is defined as an assertion that is intended to be true of every object of a class before and after every method call. It can be coded as a suitably named boolean method of the same class. For example, in a class that maintains a private list of objects, the invariant could state the implementer's intention that the list should always be circular. While the program is under test, the invariant can be retested after each method call, or even before as well.

9. Invariants

Invariants are widely used today in software engineering practice, though not under the same name. For example, every time a PC is switched on, or a new application is launched, invariants are used to check the integrity of the current environment and of the data held in long-term storage. In Microsoft Office, invariants on the structure of dynamically allocate storage on the heap are used to help diagnose storage leaks. In the telephone industry, they are used by a so-called software auditing process, which runs concurrently with the switching software in an electronic exchange. Any call records that are found to violate the invariant are simply re-initialised or even just deleted. It is rumoured that this technique once raised the reliability of a system from undeliverable to irreproachable. d.

In Microsoft, I see a future role for invariants in post-mortem dump-cracking, to check whether a failure was caused perhaps by some incident long ago that corrupted object data on the heap. Such a test has to made on the customer machine, because the heap is too voluminous to communicate the whole of it to a the central server. in Redmond.

10. Interface assertions

Assertions written at the interfaces between program modules, assemblies and components give exceptionally good value. Firstly, they are exploited at least twice, by the implementer of the interface and by its user – indeed by all its users. Secondly, interfaces are ~~usually more~~ often quite stable over successive releases than the code, so the assertions ~~that define an interface~~ are used repeatedly whenever code is enhanced for a later release. ~~This should make it safer for~~ In Fact

15 mins

should be encouraged rather

the users of a library ~~to~~ read the interface documentation than the actual code of the method itself. Interface assertions permit unit testing of each module separately from its use; and they give good guidance in the design of rigorous test cases. Finally, they enable the analysis and ~~proof~~ verification of a large system to be split into smaller parts, so that each part can be analysed separately in a modular fashion. This is absolutely critical. Even with fully modular checking, the first application of PREfix to Windows 2000 took three weeks of machine time; and even after a series of optimisations and compromises, it still takes three days.

11. Preconditions

The first important kind of assertion that one sees at an interface is a precondition. A precondition is defined as an assertion made at the beginning of a method body. It is the caller of the method rather than the implementer who is responsible for the validity of the precondition on every entry to the method; the implementer of the body of the method can just take it as an assumption. Recognition of this division of responsibility across the interface protects the virtuous writer of a method from being called out to inspect faults which have been caused by a careless caller of the method. As an example, consider the insertion of a node in a circular list, which may require that the parameter is not NULL. The example shown on this slide includes also a test of the class invariant and a simplifying assumption; the assumption uses the find method local to the same class to check that the inserted object is not already there.

12. Post-conditions

The second main kind of interface assertion is the postcondition, defined as an assertion evaluated on return from the method. The postcondition ~~is an assertion which~~ describes (at least partially) the purpose of ~~a~~ the method~~call~~. The caller of a method is allowed to assume its validity on return from the call. The obligation is on the writer of the method to ensure that the post-condition is always satisfied, and that the class invariant is satisfied too. Preconditions and post-conditions document the contract between the implementer and the user of the methods of a class. The contractual aspect of assertions has been heavily exploited in the Eiffel programming language.

13. ASSERTIONAL

It is sometimes useful for an assertion to refer to ~~the previous~~ earlier values of variables, or to a log of significant actions that a program has previously performed. That is the purpose of the assertional macro, which may contain arbitrary declarations of variables, and assignments to them. The example on this slide shows the use of an assertional variable to hold the initial value of a real variable t, so as to check in the postcondition that its value has been reduced by the body of a method.

14. Optimisation

Assertions can help a compiler produce better code. For example, in a C-style case statement, a default clause that cannot be reached should marked with an UNREACHABLE assertion, and the compiler avoids ~~emission of~~ generating unnecessary code for this case. In future, perhaps assertions will give further help in optimisation, for example by asserting that two pointers or references do not point to the same location. This will encourage the compiler to continue optimisation, in spite of the risk of an alias clash. The optimisation

20 min

of course by

depends on confidence in the validity of the assertion. At present this confidence is built up ~~on~~ massive testing – in fact, assertions are widely believed to be the only reliable form of program documentation. When assertions are automatically proved by an analysis tool, they will be even more believable.

3

15. Defect tracking

Assertions feature strongly in the code for Microsoft Office – around a quarter of a million of them. They are automatically given unique tags, so that they can be tracked in successive tests, builds and releases of the product, even though their line-number changes with the program code. Office Watson automatically records and classifies assertion violations in RAID. When the same fault is detected by two different test cases, it is twice as easy to diagnose, and twice as valuable to correct. This kind of fault classification defines an important part of the team's programming process.

In future, defect tracking will be assisted by the distinction between preconditions and postconditions. Violation of a precondition will be attributed to the calling program, whereas violation of a postcondition or invariant will be attributed to the called method.

16. PREFIX ASSUME

The global program analysis tool PREfix is now widely used by Microsoft development teams t~~o detect program defects at an early stage, even before testing. Typical defects are a NULL pointer reference, an array subscript out of bound, a variable not initialised.~~ PREfix works by analysing all paths through each method body, and it gives a report for each path on which there

control

5 min
2 pages

may be a defect. The trouble is that most of the paths considered can never in fact be activated. The resulting false positive messages still require considerable effort to analyse and reject; and the rejection is prone to error too, as shown in the case of the nimda virus.

Assertions can help the PREfix a~~nomaly checke~~r to avoid unnecessary noise. If something has only just three lines ago been inserted in a table, it is annoying to be told that it might not be there. The ASSUME macro allows the programmer to tell PREfix information about the program that cannot be automatically deduced. This is a much better way of reducing noise than just switching off the warning.

17. SHIP-ASSERTS

The original purpose of assertions was ~~to~~ preferably ensure that program defects are detected as early as possible in test, rather than later-on, after check-in, after code complete, or even after delivery. But the power of the typical customer's processor is constantly increasing, and the frequency of delivery of software upgrades in the dot.NET environment is also increasing. It is therefore more and more cost-effective to leave an increasing ~~certain~~ proportion of the assertions in the retail code; when they fire they generate an exception, and the choice is offered to the customer of sending a bug report to Microsoft. This is much better than a crash, which is a likely result of entry into a region of code that you already know has never been encountered in test. A common idiom is to give the programmer control over such a range of options by means of different ASSERT macros. These three examples are taken from the Visual Studio project. In libraries provided by Microsoft to its customers, most of the preconditions will be SHIP-ASSERTS like this.

25 min

18. Life of an assertion

In this talk, I have described many effective ways in which assertions of various kinds are exploited today in Microsoft programming practice. The~~se~~ir benefits extend through all stages of the ~~Microsoft~~ software development process. My final suggestion is that the benefits can be increased, and the costs reduced, if the same assertion can be reused for more than one purpose during the design, development and deployment and evolution of Microsoft products.

To maximise the benefit of assertions, start to write them even before the code into which they will be inserted. ~~Then~~ At this stage, they help in early design discussions, clarifying the design options, and enabling ~~their~~ consequences of the alternative choices to be evaluated. In project planning, use assertions as interface contracts, formalising the assumptions and commitments of each separately developed module of code. They are a powerful tool to control unavoidable dependencies that plague the life of product managers. Exploit the assertions again in the planning of exhaustive test strategies, and incorporate them directly into early test harness designs. Make sure that the tests that violate assertions are given priority, and that they are included in long-term regression suites. In detailed coding, ~~checkable~~ assertions record the reasons why the program is believed to work, and this reasoning can be checked in code reviews; ~~and this goal~~ assertions actually make~~s~~ the review more interesting and more effective. In early prototyping of new features, a simplifying assumption gives a safe way of recording future obligations undertaken by the programmer.

19.... continued

In the debug phase of development, add a new assertion that will detect possible recurrence of each error that has been detected ~~occurred~~. Classify defects in RAID according to the assertions that they fire. Before RTM, decide which assertions to leave in ship code, and what should be the appropriate logging or recovery action. Incorporate invariants and other assertions into ~~code~~ the code that is deployed for for start-up checks, for software audit, and for dump analysis and diagnosis. Finally, exploit and strengthen the assertions that you already find in legacy code, so that their valuable role is repeated again and again when the code is evolved for subsequent releases.

20. Conclusion

In this lecture I have
~~As~~ I ~~have just~~ described, ~~there are~~ so many ways of exploiting an assertion during its lifetime that there is no need to know in advance which of the uses will be most valuable in each case. Once an assertion has been recorded, it can be used and re-used to meet the evolving needs of the project. In conclusion, my message is simple:

Assert early, assert often, and assert more strongly every time.

21. Apologies to…

My title is taken from the advice that I was given when I first voted in an election in Ireland in 1970. It is attributed to a nineteenth century American humourist, Josh Billings.

22. Acknowledgements

30 min

Apologies are due to Josh Billings, and more serious acknowledgement to all who have responded to my earlier research survey on the current uses of assertions in Microsoft, ~~and those who encouraged and enabled me to make it~~. I would like to continue to collect more experience of the use of assertions in other environments. Further contributions will be very welcome if sent to thoare@microsoft.com. I believe that these contributions from the best of Microsoft Development practice **today** will be of enormous value in the planning and design of **future** programmer productivity tools, which will ~~make further major contributions to efficient~~ delivery of highly trustworthy software products from Microsoft.

exploit your assertions even more effectively to ensure ~~continued~~ and efficient

……………………………………to be continued, and probably shortened significantly. Remarks about the need to restrict the language in which probes are written, and/or the desirability of extending it. analysis to verify innocense of probes.

Conclusion. the role of C#

See the appendix for a summary of the multiple uses of assertions

~~There is no doubt that~~ An ~~an~~ excellent way of propagating ~~the good~~ practices that lead to trustworthy code is the use of programming language features that are inherently safe and promote correctness. That is why our research team recently welcomed a ~~challenge from~~ assignment the C# team to propose a design of an assertional feature for their language, which lacks the macro capability that has been used to add assertions to C and C++. C# provides an exceptionally good handle on which to hang the assertions in their role as probes, namely the **interface** class. We hope to get agreement from the C# design authority to put some simpler forms of assertions onto the interface. Resource constraints over the next two years prevent the consideration of the ~~a~~ more ambitious ideas ~~proposal~~, ~~along the lines~~ suggested in this paper.

But this does not deter us from the development of toolsets which will be initially ~~thoroughly~~ tried and tested by application to the existing Microsoft code base, written in legacy languages. The lessons learned can then be transferred with confidence to the new language, when the opportunity again arises. managed subset of C#

The concept of managed code in C# gives an excellent example; ~~establis~~ it establishes trust that code will do nothing bad. Assertions state a more positive form of trust; that something good reasoning can be checked in code reviews; and this goal makes the review more interesting and more effective. In early prototyping of new features, a simplifying assumption gives a safe way of recording future obligations undertaken by the programmer.

In the debug phase of development, add a new assertion that will detect possible recurrence of each error that has occurred. Classify defects in RAID according to the assertions that they fire. Before RTM, decide which assertions to leave in ship code, and what should be the appropriate logging or recovery action. Incorporate invariants and other assertions into code for start-up checks, software audit, and dump analysis and diagnosis. Finally, exploit and strengthen the assertions that you already find in legacy code, so that their valuable role is repeated again and again when the code is evolved for subsequent releases.

In fact, there are so many ways of exploiting an assertion during its lifetime that there is no need to know in advance which of the uses will be most valuable in each case. Once an assertion has been recorded, it can be used and re-used to meet the evolving needs of the project.

Acknowledgements