Static Code Analysis

Code review

- Code review is one of the oldest and safest methods of defect detection.
- It deals with joint attentive reading of the source code and giving recommendations on how to improve it.
- Reveals errors or code fragments that can become errors in future.
- Analysis without executing code

Static Code Analysis

- Static code analysis is the process of detecting errors and defects in software's source code.
- Static analysis can be viewed as an automated code review process.
- Clears main disadvantage of joint code review – high cost

Static vs Dynamic

Static Analysis

- Examine code
- Handles unfinished code
- Can find backdoors
- Potentially complete

Dynamic Analysis

- Run code
- Code not needed, eg, embedded systems
- Has few(er) assumptions
- Covers end-to-end or system tests

Tasks solved by SCA

- Detecting errors in programs
- Recommendations on code formatting
- Metrics computation

Different Static Analyzers Are Used For Different Purposes

- To check intellectual property violation
- By developers to decide if anything needs to be fixed (and learn better practices)
- By auditors or reviewer to decide if it is good enough for use

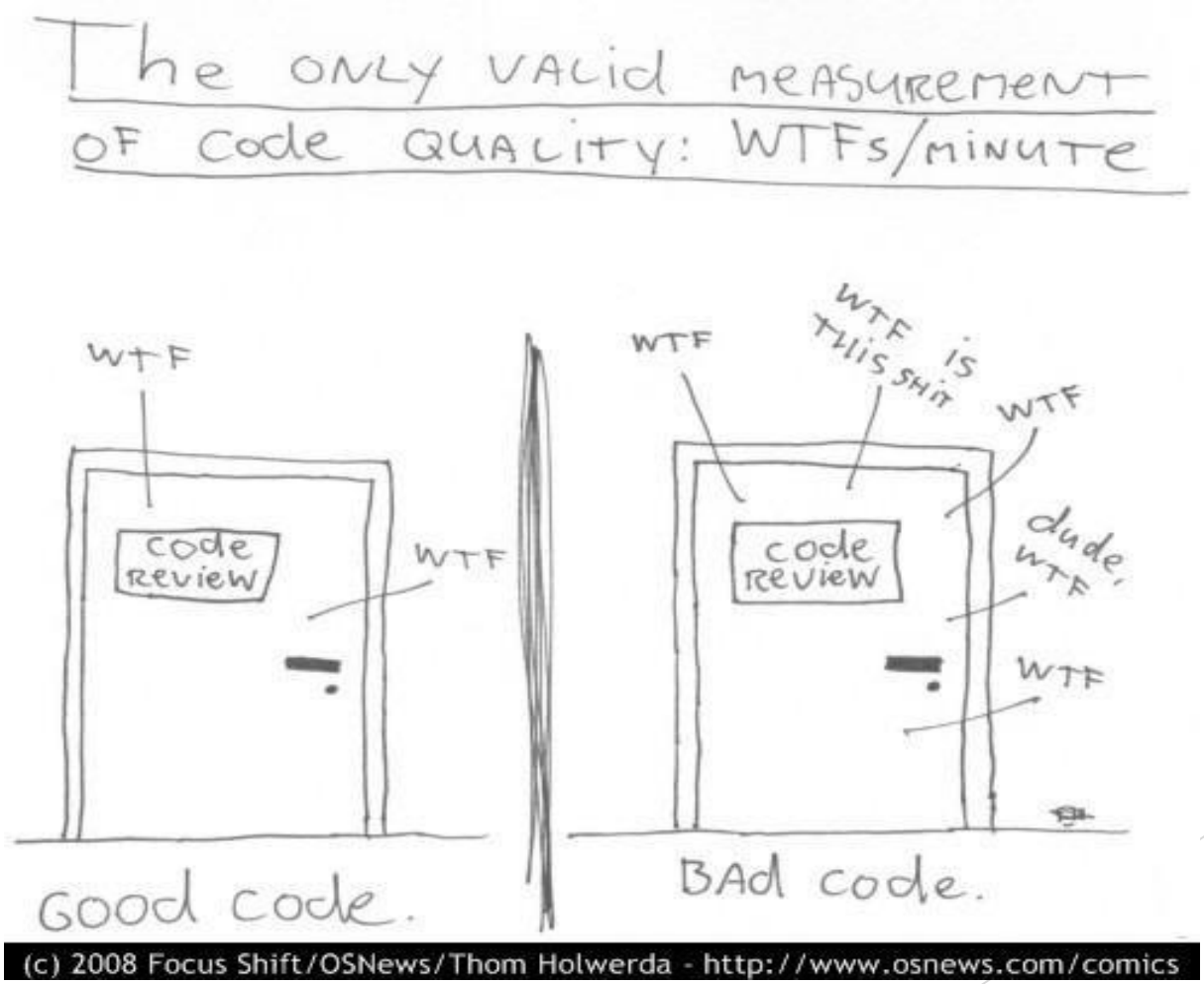
Why?

- One of a collection of strategies for improving code quality
- Code compliance to company wide standard
- Identify (potential) bugs in code - Identify potential issues earlier in development cycle
- Identify design and implementation problems
- Peer education

Other benefits

- Full code coverage (code fragments that get control very rarely)
- Doesn't depend on the compiler
- Doesn't depend on the environment
- Find ghost consequences of Copy-Paste usage

Increase code quality i.e. reduces WFT/min ratio



StyleCop

- Static Code Analysis - C# only
- Written by Microsoft
- Different than FxCop
- Free (as in speech)
- Not built in to Visual Studio
- Enforces style guidelines

Rules

- Documentation
- Layout
- Maintainability
- Naming
- Ordering
- Readability
- Spacing

FxCop

FxCop is a code analysis tool that checks .NET managed code assemblies for conformance to the Microsoft .NET Framework Design Guidelines. It uses reflection, MSIL parsing, and callgraph analysis to inspect assemblies for more than 200 defects in the following areas:

- Library design
- Localization
- Naming conventions
- Performance
- Security

Static code analysis – fear of change

- We already do code reviews
- Way too many rules
- Not clear what rules to use
- We must have different rules
- Too many violations to fix
- Who’s going to fix the violations?
- Hindrance to creativity
- Yet another bureaucratic invention

Implementing static code analysis

- Identifying appropriate rules
- Handling backlog
- Setting up the process
- Educating the team
- Staying agile!

Disadvantages

- Static analysis is usually poor regarding diagnosing memory leaks
- Hard to find concurrency errors (execute a part of the program)
- False-positive reports

- Dynamic analysis helps

Writing Clean Code

Why Write Clean Code?

- Writing clean code reduces number of defects.
- Writing clean code reduces cost of change.
- Clean code is easier to learn.
- Writing clean code helps manage complexity (keeps code complexity to a minimum).

Managing Complexity

- In terms of software development, there are two different types of complexity:
  - **Essential Complexity** – Revolves around the fact that software is intrinsically complex, and no methodology or tool is going to eliminate this complexity.
  - **Accidental Complexity** – Occurs due to a mismatch of paradigms, methodologies, and/or tools in our application.
- Solution:
  - Minimize the amount of essential complexity that anyone's brain has to deal with at any one time.
  - Keep accidental complexity from needlessly proliferating.

Always Keep in Mind

- Our primary audience are fellow humans, not computers.
- Bad code breeds more bad code.
- Design as if designing from scratch, never pick a solution that is easiest to implement, but the solution easiest to maintain.
- Errors should be detected as soon as possible.
- LeBlanc's law: Later equals never.

Avoid Copy and Paste

- Avoid copy and paste (duplicating) while you are writing code.
  - Duplicate code requires parallel modifications.
  - If you are using copy and paste while you're coding, you are probably committing a design error.
  - It also violates the DRY (Don't Repeat Yourself) principle.

Avoid Methods that do More Than One Task

- Avoid writing methods that do more than one task. Those methods can be split into multiple smaller methods or even classes.
  - Long methods (200+) and methods with long argument lists (5+) usually do more than one thing.
  - Writing methods that do more than one task lowers readability of code.
  - It is hard to reuse them, which often leads to duplicated code.
  - Writing methods that do more than one task increase accidental complexity.
  - These methods are often victims of additional improvements.

Avoid Deeply Nested Control Structures

- Avoid writing deeply nested control structures.
  - Poor use of control structures increases complexity; good use decreases it.
  - How it is often used:

```
if (firstCondition)
{
     if(secondCondition)
     {

     }
}
```

  - How it should be used:

```
if (firstCondition && secondCondition)
{
}
```

Avoid Returning Null

- Think twice before returning null.
- When writing a method, never return null for collections and arrays, return empty arrays and collections instead.
- In case when returning an interface, consider returning a default implementation instead of null.
  - How it should be used:

    Let's say we have a software that provides a list of documents. The list of documents can be filtered, and the last used filter is stored for next use. Filter is defined with a following interface:

```
interface IDocumentFilter
{
    List<Document> Filter(List<Document> documents);
}
```

Avoid Returning Null

- How it should be used:

When a user starts the application last used filter is restored with this code:

```
IDocumentFilter filter = filterStore.GetLastFilter();
```

Instead of returning null when there is no last used filter, consider returning something like this:

```
class NoFilter : IDocumentFilter
{
    public List<Document> Filter (List<Document>
documents)
    {
        return documents;
    }
}
```

With this approach `filterStore.GetLastFilter()` users do not have to worry about handling nulls.

Do not Spread Object Logic over Multiple Classes

- Object logic should not be spread over multiple classes. Whenever you do something complex with object's attributes and/or methods ask yourself if these should be object's method.
  - Not spreading object logic over multiple classes allows better code reuse.
  - How it is often used:

```
public double CalculateCost (Shape shape)
{
     return shape.Width * shape.Height * costPerArea;
}
```

  - How it should be used:

```
public double CalculateCost (Shape shape)
{
     return shape.Area * costPerArea;
}
```

Use Assertions

- Use asserts to verify conditions that should never be false (sanity checks, argument checks in internal/private methods). Asserts are not present in the release build.
  - Assert checks for a condition; if the condition is false, outputs a specified message and displays a message box that shows the call stack.

```
System.Debug.Assert(bool condition, string message)
```

  - How it should be used:

```
public void CloseAll()
{
    foreach(var window in Windows)
    {
        window.Close();
    }
    Debug.Assert(Windows.Count == 0, "All windows should
be closed");
}
```

Check Method Arguments

- Check method arguments for validity for all public methods.
- Throw appropriate exceptions if arguments are not valid (*ArgumentException* and its subclasses).
- Use assertions to verify argument validity for internal/private methods.
- An important exception is the case in which the validity check would be expensive or impractical and the validity check is performed implicitly in the process of doing the computation.

Be Careful When Casting With “as”

- Syntax: animal as Cat vs. (Cat)animal
- Use “as” cast only when you have already checked type or when dealing with failed cast uses the same code path as dealing with null value (implementing Equals method).
  - Perfect for error propagation.
  - ”As” cast returns null when cast fails, regular cast throws an exception.

Throwing Exceptions

- Always throw correct exceptions. Never use pure *Exception*. Favor the use of standard exceptions (ie *ArgumentException*, *ArgumentNullException*, *NotSupportedException*). Define new exception types in order to hide implementation details.
  - How it should be used:

    GetUser method should throw *UserNotFoundException*, not *SQLException* or *FileNotFoundException*.
- Don't leave the object in an inconsistent state when throwing exceptions.

Throwing Exceptions

- Mind the callstack when re-throwing exceptions. When you must explicitly re-throw exception wrap it in another exception.
  - How it is often used:

```
catch(Exception e)
{
    throw e; // This overwrites callstack, use just throw
}
```

  - How it should be used:

```
catch(Exception e)
{
    throw; // This overwrites callstack, use just throw
}
```


Writing Optimizd Code

General Tips

- The model will someday be much bigger than the one used on product.
- Will your part of the code work with millions of objects, or just a couple of them?
- Will your part of the code work on one machine or in a dual system?
- What systems will it run on in the enterprise environment?
- If you do not know how fast some part of the code is, make a test and run it a million times. Then try some other code that solves the same issue, and compare the measured results.

General Tips

- When writing a loop, ask yourself how many times it will usually execute until exit? Is the code in the loop optimal?
- Logging is slow. Log only important information. If preparing data for logging is time consuming, check first if that log level is on.
- Avoid creation of unnecessary objects (an empty object consumes 16 bytes)
- Take care of the memory alignment of structure attributes.
- Do not repack collections if it can be avoided.
- Do not create a class attribute if it is used as a local variable.

General Tips

- Instantiation of a large number of empty objects, especially collections, should be avoided. In that case, null value of the objects should be properly handled.
- If it is necessary to have an object as the class attribute temporarily, try to clear it and set it to null as soon as it is no longer needed.
- If you want to copy some code from another component, ask yourself if that part of the code can be placed somewhere in the common component?
- Try to use common code as much as possible.

List Collection

- Consider using basic array instead of List object.
  - Using a list object makes an overhead of 40 bytes per instance over an array and makes an overhead of double capacity check when an entity is added to the list.
- Create a list object with predefined capacity whenever possible.
  - By default a list object is initialized with a capacity of 0. When the first item is added, it is reinitialized to a capacity of 4. Subsequently, whenever the capacity is reached, the capacity is doubled.
  - How it is often used:

```
List<int> listObj = new List<int>();
```

  - How it should be used:

```
List<int> listObj = new List<int>(157345);
```

- If list members are known up front, it is better to calculate the capacity up front than to create an empty list and then populate it.

List Collection

- If the capacity of the list is not known up front, it should not be instantiated with capacity of 0. Instead, the list should be instantiated using the default constructor.
  - How it is often used:

```
List<ModelCode> properties = new List<ModelCode>(0);
```

  - How it should be used:

```
List<ModelCode> properties = new List<ModelCode>();
```

- If the capacity of the list is not known up front and elements of the list are rarely added or removed, compact of the list should be done after adding or removing objects, in order to decrease the capacity of the list to the number of objects that are stored in the list.
  - How it should be used:

```
properties = new List<ModelCode>(properties);
```

List *Contains* Method

- Use list *Contains* method only when you are 100% sure that it is always going to be a small collection. Use HashSet instead to optimize searching for an element in a list.
  - How it is often used:

```
List<int> listObject = new List<int>(1300154);
. . .
for (. . .)
{
      if(listObject.Contains(...))
      . . .
}
```

  - How it should be used:

```
List<int> listObject = new List<int>(1300154);
. . .
HashSet<int> hsObject = new HashSet<int>(listObject);
for (. . .)
{
      if(hsObject.Contains(...))
      . . .
}
```

Threads

- Do not create a thread for everything. Ask yourself if the parallel work will execute continuously, periodically or time based?
- Use thread pool threads for some occasional jobs. Be aware that getting a thread from the thread pool can take some time. Sometimes even a few tens of seconds.
- Use Timers for time based triggered jobs. This will be executed in the background in a thread taken from the thread pool. Triggering a timer can be delayed.
- Use .Net Tasks
  - Easy to use.
  - Manages task scheduling.
  - Manages exceptions that tasks throw.
- Each thread demands some resources. If known up front all that a thread will do, setup the thread stack size.
  - Default stack size for x64 applications is 4MB. Usually, for small threads, 256k is enough.

Code Review Checklist

Code Review Checklist

- In order to ensure that given best coding practices are followed and quality criteria is satisfied, "**Code Review Checklist Template**" document can be defined
- It is comprised of **inspection items** mapped to prescribed best coding practices
- As a result of code review, checklist in this document should be completed by a Reviewer.
- It is mandatory that the Reviewer:
  - Fill in basic information about code review (chapter "Code review details"),
  - Fill in "Status" field in "Code review checklist" chapter of this document for every inspection item.
    - It is **mandatory** that the "Status" field in "Code review checklist" chapter of this document be updated for every one of the inspection items;
    - No blank fields in this column are permitted.

Code Review Checklist

- "Status" field value should be set in accordance with following rules:
  - **N/A** – Inspection item is **not found** in the source code during code review.
  - **OK** – Inspection item is **found** in the source code during code review and it is **properly implemented** (according to guidelines in "How to write optimized and clean code" document).
  - **Issues found** – Inspection item is **found** in the source code during code review, but it is **not properly implemented**.
  - **Issues corrected** – Inspection item is **found** in the source code during code review and it was **not properly implemented**, but it is **corrected** during code review.

Where to Go Next

- McConnell, Steve. *Code complete. Redmond*, Wash: Microsoft Press, 2004. Print.
- Bloch, Joshua. *Effective Java. Upper Saddle River, NJ: Addison-Wesley, 2008. Print.*
- Martin, Robert C. *Clean code : a handbook of agile software craftsmanship. Upper Saddle River, NJ: Prentice Hall*, 2009. Print.

Thank you for your attention