Guia do Desenvolvedor

Prodesp
MCTQ/GPQ

Fevereiro/2016

Versão 1.2

CONTROLE DE REVISÃO

Data	Versão	Descrição da revisão	Responsável
17/1/2013	Draft 1.0	Elaboração inicial.	Sibinel
23/1/2013	Draft 1.1	Ajustes de layout	Sibinel
04/02/2013	Draft 1.2	Elementos de Boas Práticas	Sibinel
23/02/2016	1.0	Adequação à Metodologia da Prodesp	Cindy
04/03/2016	1.1	Inclusão da Aplicação de Referência em Java	Cindy
11/03/2016	1.2	Inclusão da Aplicação de Referência em .NET e Métricas nDepend	João Boaventura

Sumário

1. Introdução

Este documento tem como objetivo elencar os padrões gerais para construção de aplicações utilizando as tecnologias/linguagens .NET/C# e Java. O documento apresenta elementos para orientação designados pela Microsoft, manuais de boas práticas de mercado e pela comunidade Java. É composto por elementos relacionados à Codificação e Tecnologia.

2. Codificação

Aplicação de Referência

A Prodesp disponibiliza para consulta e utilização aplicações de referência em Java e .NET para projetos de média complexidade. Os códigos-fonte encontram-se no StarTeam, projeto "Prodesp – Arquitetura Referencia". Para acesso, favor enviar um Notes para ctq@sp.gov.br solicitando acesso de leitura ao mesmo.

Link para a aplicação em Java:

http://10.200.45.234/referencia/

Link para a aplicação em .NET:

http://10.200.142.177/ReferenciaNET

Termos em Português

Todos os termos do projeto devem ser utilizados em Português, ou seja, a Solution, os projetos, os namespaces, classes, métodos parâmetros, variáveis, etc.

Abreviações

Evitar o uso de abreviações para os nomes dos identificadores ou verbos que tenham escopo público. Abreviações dificultam o entendimento do código. Para o escopo local, de preferência para nomes mais sintéticos, que não poluem o método. Por exemplo:

Tipo	Correto	Errado
Indice Local	i ou idx	indiceDeElementos
Método Público	BuscarPedidoPorData	BPedidos
Método Privado	ValidarData	ValidarDataEntregaPedidoNula

Consistência

Seja consistente com a nomenclatura, mostrando os relacionamentos e destacando as diferenças. Segue exemplo das notações abaixo:

Errada

```
public class EstoqueQueue
{
    public int NoDeElementosQ { get; set; }
    public int PrimeiroElementoQueue { get; set; }
    public int QueueCapacidade { get; set; }

    public void IncluirQ(Item item)
    {
    ...
}
```

C o r r e t a

```
public class EstoqueQueue
{
    public int NoDeItens { get; set; }
    public int PrimeiroItem { get; set; }
    public int Capacidade { get; set; }

    public void Incluir(Item item)
    {
    ...
}
```

Observe abaixo que a notação **C o r r e t a** fica mais clara:

```
    public void Doit()
    {
        var queue = new EstoqueQueue();
        Console.WriteLine("Capacidade: {0}", queue.Capacidade);

        queue.Incluir(new Item
        {
            Nome = "caneta",
            Quantidade = 10
        });

        Console.WriteLine("Qtd: {0}", queue.NoDeItens);
    }
```

Não usar notação Húngara em hipótese alguma

A notação Húngara esta descontinuada pela própria Microsoft, e não deve ser usada no Projeto. Também não deve ser usado underscore ('_') ou abreviações.

Legibilidade

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

-Martin Fowler et al, Refactoring: Improving the Design of Existing Code, 1999

Escreva o código primeiro para as pessoas, depois para o computador. O tempo gasto para escrever código legível é o mesmo para escrever código confuso. A regra fundamental é: "Use, mas não abuse".

Nomenclatura auto-explicativa permite a compreensão do código sem a necessidade de comentários. Por isso, evite o uso de nomes genéricos em classes concretas.

Outro tipo de construção que afeta a legibilidade é a construção de expressões complexas e, portanto, deve ser evitada.

Coesão e Acoplamento

O termo coesão se refere ao grau de relacionamento entre as operações de um determinado método. Isto significa que quanto maior o grau de relacionamento entre as operações do método, maior o grau de coesão do método, indicando alta qualidade de codificação. Um método de cálculo de seno (Sin(...)) trata somente deste cálculo, enquanto que um método para cálculo de seno e tangente (SinTan(...)) contem uma complexidade extra para tratamento das duas operações. Estudos mostram que métodos com alto grau de coesão apresentam um percentual menor de falhas.

Níveis de Coesão Aceitáveis

Coesão Funcional: Este tipo de coesão é o mais forte e o melhor tipo, pois se encaixa na própria definição do conceito. Ela ocorre quando o método executa somente uma operação ou função. Evidentemente ele deve estar adequadamente nomeado para tal.

Coesão Sequencial: Este tipo de coesão ocorre quando o método contem operações que devem ser executadas em uma ordem específica, além de compartilhar dados entre os passos executados. Um exemplo típico é quando um método deve acessar arquivos em disco e o processo de abertura, leitura de dados e fechamento deve acontecer sempre na mesma ordem. A maioria das vezes é possível refatorar e transformar coesão temporal em funcional.

Coesão Temporal: Coesão temporal refere-se a métodos que tem uma agenda especifica de execução, ou tratamento. São exemplos típicos, *Startup()*, *Shutdown()*, *Configure()*, etc.

Níveis de Coesão Inaceitáveis

Coesão Procedural: Esse tipo de coesão ocorre quando os elementos são associados conforme seus relacionamentos procedurais ou algorítmicos. Sem o contexto da aplicação, os módulos parecem estranhos e muito difíceis de entender. A solução consiste em reprojetar o sistema, utilizando OO.

Coesão Lógica: Esse tipo de coesão ocorre quando um método apresenta um conjunto de funções relacionadas e uma delas é escolhida através de um parâmetro de controle em sua chamada. A solução é refatorar o método, criando métodos diferentes para cada tipo de execução.

Coesão Caótica:

TRATAR "WARNINGS" COMO "ERRORS"

Uma estratégia interessante é ajustar a configuração do Visual Studio para tratar todos os "Warnings" como "Errors". Este modelo obriga o desenvolvedor a resolver os problemas antes de continuar a codificação.

OS VERBOS DOS MÉTODOS DEVEM ESTAR NO INFINITIVO.

Na criação dos métodos, devem ser usados verbos ativos e estes verbos devem estar no infinitivo. Exemplo:

```
 public class PessoaRepo : Repositorio
{
    public void Persistir(PessoaEntity pessoa)
    {
        CurrentContext().Set<PessoaEntity>().Add(pessoa);
        Console.WriteLine("Persistir: " + pessoa.ToString());
    }
}
```

NÃO USAR PLURAL.

Os substantivos (nomes) devem sempre estar no singular, tanto quando estão sendo aplicados a tipos, quando compostos em verbos dos métodos. O plural somente pode ser usado para nomes de propriedades do tipo lista (IList<?>).

```
        /// <summary>
        /// Lista as pessoas que tem o nome iniciado pela string de entrada (like)
        /// </summary>
        /// <param name="nome">Nome para busca</param>
        /// <returns>Lista de pessoas encontradas</returns>
        public IList<PessoaEntity> ListarPessoa(string nome)
        {
            try
            {
                return PessoaRepo.ListarPorNome(nome);
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
                throw;
            }
        }
```

```
    public class PessoaEntity
    {
        public short Id { get; set; }
        public string Nome { get; set; }
        public string Endereco { get; set; }
        public IList<string> Apelidos { get; set; }
    }
```

NÚMEROS MÁGICOS

Números mágicos são todas as constantes, tamanhos de vetores, fatores de conversão, ou qualquer literal expressa no código. O seu uso deve ser evitado dentro do corpo de métodos, getters, setters e delegates. Todo número mágico deve ser representado através de enum ou propriedades que associem um nome ao número. Por Exemplo:

```
    public enum Operacao : int
    {
        READ = 0,
        WRITE = 1,
        UPDATE = 2,
        DELETE = 3
    }
```

```
    public class Evento
    {
        private static readonly int MaxDias = 10;

        private static readonly int MinDias = 3;

    ...
    }
```

2.1. Métricas de código utilizadas em código fonte

No processo de Integração Contínua, as ferramentas de análise estática de código são previamente parametrizadas para atender às boas práticas de programação preconizadas pelas comunidades de desenvolvedores, além de apontar bugs e violações.

Por isso, a recomendação é utilizar as ferramentas de análise estática corporativas, sem custo adicional para as equipes. Para sistemas em Java a Prodesp fornece a análise utilizando o Sonar e para sistemas em .Net a análise é realizada através do NDepend.

2.2. Convenção da Capitalização.

2.2.1. .Net/C#

A seguir é mostrada uma tabela de correspondência para o uso da capitalização:

Identificador	**Capitalização**	**Exemplo**
Class	Pascal	**PessoaBC**
Enumeration type	Pascal	**OperacaoTipo**
Enumeration values	Pascal	**TipoInclusao**
Event	Pascal	**ValorChanged**
Exception class	Pascal	**RepositorioException**
Read-only static field	Pascal	**MaxNivel**
Interface	Pascal	**IRepositorio**
Method	Pascal	**ListarPorNome**
Namespace	Pascal	**Prodesp.Seguranca**
Parameter	Camel	**enderecoTrabalho**
Property	Pascal	**PessoaRepo**
Local variable	Camel	**nomePai**
Field	Camel	**idxEndereco**
Region	Pascal	**Cadastro**

2.2.2. Java

A seguir é mostrada uma tabela de correspondência para o uso da capitalização:

Identificador	Capitalização	Exemplo
Class	Pascal	**PessoaBC**
Enumeration type	Pascal	**OperacaoTipo**
Enumeration values	ALL_UPPER_CASE	**TIPO_INCLUSAO**
Event	Pascal	**ValorChanged**
Exception class	Pascal	**RepositorioException**
Read-only static field	ALL_UPPER_CASE	**MAX_NIVEL**
Interface	Pascal	**IRepositorio**
Method	Camel	**listarPorNome**
Namespace	Pascal	**prodesp.seguranca**
Parameter	Camel	**enderecoTrabalho**
Property	Camel	**pessoaRepo**
Local variable	Camel	**nomePai**
Field	Camel	**idxEndereco**

2.3. Assemblies

O nome do projeto e por consequência o nome do Assembly devem ter sua formação guiada por uma das formas a seguir:

2.3.1. .Net/C#

Nome do Projeto	Exemplo
<Projeto>.<Camada>.dll	EFSample.Negocio.dll
<Projeto>.<Camada><TipoComponente>.dll	CDHU.Habilitacao.Persistencia.dll

<Cliente>.<Sistema>.<Camada>.dll	Ses.GSnet.Compras.Dominio.dll

2.3.2. Java

Nome do Projeto	Exemplo
<projeto>.<camada>.jar	efSample.negocio.jar
<projeto>.<camada><TipoComponente>.jar	cdhu.habilitacao.persistencia.jar
<cliente>.<sistema>.<camada>.jar	ses.gsnet.compras.dominio.jar

2.4. Namespaces

A criação do Namespace das classes, interfaces e enum deve obrigatoriamente acompanhar a nomenclatura do Assembly, ou seja, o namespace deve iniciar com o nome do assembly. A partir daí deve agregar outros componentes ou divisões necessárias. Para cada novo nó na estrutura do namespace deve ser criada uma pasta para conter os arquivos específicos deste namespace. Assim temos:

2.4.1. .Net/C#

Projeto	Namespace
EFSample.Negocio.dll	EFSample.Negocio.Util
CDHU.Habilitacao.Persistencia.dll	CDHU.Habilitacao.Persistencia.Mapeamento

2.4.2. Java

Projeto	Namespace
efSample.negocio.jar	efsample.negocio.util
cDHU.habilitacao.persistencia.jar	cdhu.habilitacao.persistencia.mapeamento

2.5. Classes

Para a criação de classes devem ser seguidas as seguintes regras:

- Criar somente uma classe por arquivo fonte (.cs ou .java).
- Não usar prefixo
- Usar sufixos de acordo com o tipo de classe, conforme a seguir:

Sufixo	**Tipo da classe**	**Exmplo**
Entity	Classes de domínio	PatioEntity
Repo	Classe especifica de repositório	PlanilhaRepo
BC	Classe de negócio	VeiculoBC
Helper	Classe de apoio (library)	CriptografiaHelper
Exception	Classe de tratamento de exceção	RepoException

```
public class PessoaEntity
{
    public int Id { get; set; }

    public string Nome { get; set; }

    public string Endereco { get; set; }

    public DateTime DataInicio { get; set; }

    public virtual IList<ApelidoEntity> Apelidos { get; set; }

    public PessoaEntity()
    {
        Apelidos = new List<ApelidoEntity>();
    }
}
```

```
public class PessoaRepo : Repositorio
{
    private static Logger logger = LogManager.GetLogger("PessoaRepo");

    [Dependency]
    protected RootContext Context { get; set; }

    public void Persistir(PessoaEntity pessoa)
    {
        Register(Context);
        Context.Pessoas.Add(pessoa);
    }
    public IList<PessoaEntity> ListarPorNome(string nome)
    {
        var query = from p in Context.Pessoas
                    where p.Nome == nome
                    select p;
        return query.Take(10).ToList();
    }
}
```

```
    public class CadastroBC
    {
        [Dependency]
        protected PessoaRepo PessoaRepo { get; set; }

        /// <summary>
        /// Lista as pessoas que tem o nome iniciado pela string de entrada (like)
        /// </summary>
        /// <param name="nome">Nome para busca</param>
        /// <returns>Lista de pessoas encontradas</returns>
        public IList<PessoaEntity> ListarPessoa(string nome)
        {
            return PessoaRepo.ListarPorNome(nome);
        }
    }
```

2.6. Interfaces

Para a criação de interfaces devem ser seguidas as seguintes regras:

- Usar o prefixo 'I'
- Usar sufixos de acordo com o tipo da classe representada pela interface.

Exemplos:

Implementação	**Tipo**	**Interface**
PlanilhaRepo	Classe específica de repositório	IPlanilhaRepo
VeiculoBC	Classe de negócio	IVeiculoBC
CriptografiaHelper	Classe de apoio (library)	ICriptografiaHelper

2.7. Enumerações

ENUMERAÇÃO DE ENTIDADES DISCRETAS ....

Não usar prefixos

Usar sufixo 'EntityEnum'

ENUMERAÇÃO SELETIVA

Não usar prefixos:

Usar sufixo 'Type'

2.8. Propriedades

As propriedades devem seguir a notação nova do C# (`{ get; set; }`). Devem ser declaradas no inicio da Classe. Quanto à nomenclatura, em caso de tipos complexos, verificar a possibilidade de usar o tipo como nome da propriedade. Por fim, não devemos usar sufixos ou prefixos. Uma observação importante é que devemos sempre utilizar o operador '`virtual`' para as propriedades do tipo '`IList`', pois existem questões sobre Lazy Loading que podem ser habilitadas para este tipo de dado.

```
public class EnderecoEntity : BaseEntity
{
    public EnderecoEntity()
    {
        Clientes = new List<ClienteEntity>();
    }

    public string Bairro { get; set; }

    public string Cidade { get; set; }

    public string Logradouro { get; set; }

    public virtual IList<ClienteEntity> Clientes { get; set; }
}
```

2.9. Métodos

Métodos são geralmente verbos ativos que caracterizam alguma ação sobre a classe onde estão dispostos. Evitar o uso de dois verbos para um método, pois a coesão é uma característica importante na análise da qualidade do código. Evitar o uso de verbos que não trazem significado claro, como por exemplo: *Processar*, *Executar*, *Tratar*, etc.

Geralmente não é necessário repetir o parâmetro de entrada no nome dos métodos, assim, o tipo de parâmetro é suficiente para o entendimento da funcionalidade.

Exemplo:

```
public PessoaEntity BuscarPessoa(string nome)
```

Esta regra se aplica quando o parâmetro de entrada identifica unicamente o método, mas quando isso não é possível, por exemplo, quando somente o retorno é diferente, devemos incluir um nome para diferenciar os métodos.

Para métodos de consulta, devemos seguir o seguinte padrão:

- Retorno de lista de objetos (verbo Listar):

```
public List<PessoaEntity> ListarPessoa(string nome)
```

- Retorno de um objeto único (verbo Buscar):

```
public PessoaEntity BuscarPessoa(string nome)
```

2.10. Parâmetros

Usar nomes para parâmetros que reflitam o seu significado e não o seu tipo. Evitar uso de muitos parâmetros na assinatura dos métodos. Considere o uso de DTOs.

2.11. Parâmetros para Tipo Genéricos

A utilização de Tipos Genéricos deve ser incentivada, pois torna a "tipagem" mais consistente e menos aberta a erros, pois não evita o uso de "Casting". Usar 'T' para os nomes dos Tipos Genéricos, ou quando existir mais de um tipo, usar sufixo que contenha significado para o tipo passado como parâmetros: TValor, TChave.

2.12. Delegate e Lambda

Em estudo para posterior definição.

2.13. Custom Attributes

Usar o sufixo 'Attribute'

2.14. Comentários

Não comente o obvio. Todo o comentário deve agregar algum tipo de informação que não está aparente no código, assim, comentários como os do exemplo abaixo devem ser evitados:

```
/// <summary>
/// Busca a Pessoa pelo nome
/// </summary>
/// <param name="nome">Nome da Pessoa</param>
/// <returns>Pessoa encontrada</returns>
public PessoaEntity BuscarPessoa(string nome)
```

Usar mecanismo padrão do Visual Studio '///'. Para os comentários de métodos e classes, deve ser utilizado o mecanismo padrão do Visual Studio e todos os campos gerados devem ser comentados. Por Exemplo:

```
/// <summary>
/// Busca a Pessoa pelo nome parcial (tipo like) usando nome.StartWith(...)
/// </summary>
/// <param name="nome">string parcial do nome (pelo menos 5 caracteres)</param>
/// <returns>Pessoa encontrada ou null</returns>
public PessoaEntity BuscarPessoa(string nome)
```

Comente todo identificador público. Todo identificador que for classificado como público deve ser comentado no estilo descrito acima. Este item é muito importante pois identifica mais claramente quais são as interfaces públicas das classes.

Se você mudar a assinatura, corrija o comentário. Muitas vezes mudamos a assinatura de métodos e funcionalidades de classes, gerando muita confusão porque não alteramos os respectivos comentários.

2.15. Exceções

A primeira regra e a mais importante é: Somente use "Exception" para situações excepcionais. Não use "Exception" para tratamento de regras de negócio, como por exemplo, entidade não encontrada, campos inválidos, etc.

Defina uma classe de "Exception" por camada, para facilitar o tratamento. Geralmente são usadas as seguintes denominações: RepoException, BCException e SCException, respectivamente para as camadas de Persistência, Negocio e Serviço quando esta existir.

2.16. Regions

Evite o uso de "Regions". Geralmente as "Regions" são usadas para arquivos muito grandes, o que é desaconselhado, ou para "esconder código feio", mais desaconselhável ainda. No caso de classes muito grandes, pense em refatorar e dividir responsabilidades.

3. ITENS AVALIADOS NA ANÁLISE ESTÁTICA

3.1. Sonar

A análise estática de código Java é realizada pelo Sonar e avalia as métricas a seguir. A lista atualizada dessas métricas e mais informações podem ser obtidos em:

http://10.200.45.232:9000/coding_rules#qprofile=java-sonar-way-97417|activation=true|languages=java

A lista foi obtida na ferramenta, por isso está em inglês. Qualquer dúvida, o MCTQ está disponível para ajudar.

- ".equals()" should not be used to test the values of "Atomic" classes
- "@Override" annotation should be used on any method overriding (since Java 5) or implementing (since Java 6) another one
- "BigDecimal(double)" should not be used
- "Cloneables" should implement "clone"
- "compareTo" results should not be checked for specific values
- "ConcurrentLinkedQueue.size()" should not be used
- "Double.longBitsToDouble" should not be used for "int"
- "equals(Object obj)" and "hashCode()" should be overridden in pairs
- "equals(Object obj)" should be overridden along with the "compareTo(T obj)" method
- "FIXME" tags should be handled
- "for" loop incrementers should modify the variable being tested in the loop's stop condition
- "for" loop stop conditions should be invariant
- "hashCode" and "toString" should not be called on array instances
- "HttpServletRequest.getRequestedSessionId()" should not be used
- "instanceof" operators that always return "true" or "false" should be removed
- "Iterator.hasNext()" should not call "Iterator.next()"
- "Iterator.next()" methods should throw "NoSuchElementException"
- "java.lang.Error" should not be extended
- "object == null" should be used instead of "object.equals(null)"
- "Object.finalize()" should remain protected (versus public) when overriding
- "Object.wait(...)" and "Condition.await(...)" should be called inside a "while" loop
- "Object.wait(...)" should never be called on objects that implement "java.util.concurrent.locks.Condition"
- "public static" fields should be constant
- "ResultSet.isLast()" should not be used
- "return" statements should not occur in "finally" blocks
- "runFinalizersOnExit" should not be called
- "static final" arrays should be "private"
- "StringBuilder" and "StringBuffer" should not be instantiated with a character
- "switch case" clauses should not have too many lines
- "switch" statements should end with a "default" clause
- "switch" statements should have at least 3 "case" clauses
- "switch" statements should not contain non-case labels
- "switch" statements should not have too many "case" clauses
- "TODO" tags should be handled

- "toString()" and "clone()" methods should not return null
- "wait(...)" should be used instead of "Thread.sleep(...)" when a lock is held
- "wait(...)", "notify()" and "notifyAll()" methods should only be called when a lock is obviously held on an object
- A "for" loop update clause should move the counter in the right direction
- A close curly brace should be located at the beginning of a line
- A field should not duplicate the name of its containing class
- Array designators "[]" should be located after the type in method signatures
- Array designators "[]" should be on the type, not the variable
- Assignments should not be made from within sub-expressions
- Branches should have sufficient coverage by unit tests
- Case insensitive string comparisons should be made without intermediate upper or lower casing
- Class names should comply with a naming convention
- Class variable fields should not have public accessibility
- Classes from "sun.*" packages should not be used
- Classes should not be empty
- Classes should not be too complex
- Classes that override "clone" should be "Cloneable" and call "super.clone()"
- Collapsible "if" statements should be merged
- Collection.isEmpty() should be used to test for emptiness
- Collections should not be passed as arguments to their own methods
- Collections.emptyList(), emptyMap() and emptySet() should be used instead of Collections.EMPTY_LIST, EMPTY_MAP and EMPTY_SET
- Constant names should comply with a naming convention
- Constants should not be defined in interfaces
- Control flow statements "if", "for", "while", "switch" and "try" should not be nested too deeply
- Cookies should be "secure"
- Credentials should not be hard-coded
- Cryptographic RSA algorithms should always incorporate OAEP (Optimal Asymmetric Encryption Padding)
- Declarations should use Java collection interfaces such as "List" rather than specific implementation classes such as "LinkedList"
- Deprecated code should be removed eventually
- Deprecated elements should have both the annotation and the Javadoc tag
- Empty arrays and collections should be returned instead of null
- Empty statements should be removed
- Enumeration should not be implemented
- Exception classes should be immutable
- Exception handlers should preserve the original exception
- Exception types should not be tested using "instanceof" in catch blocks
- Exceptions should not be thrown in finally blocks
- Execution of the Garbage Collector should be triggered only by the JVM
- Exit methods should not be called
- Expressions should not be too complex
- Field names should comply with a naming convention
- Fields in a "Serializable" class should either be transient or serializable
- Floating point numbers should not be tested for equality
- Future keywords should not be used as names

- Generic exceptions should never be thrown
- Generic wildcard types should not be used in return parameters
- Identical expressions should not be used on both sides of a binary operator
- IllegalMonitorStateException should not be caught
- Interface names should comply with a naming convention
- IP addresses should not be hardcoded
- Labels should not be used
- Lambdas and anonymous classes should not have too many lines
- Lamdbas containing only one statement should not nest this statement in a block
- Literal boolean values should not be used in condition expressions
- Local variable and method parameter names should comply with a naming convention
- Local Variables should not be declared and then immediately returned or thrown
- Local variables should not shadow class fields
- Loggers should be "private static final" and should share a naming convention
- Long suffix "L" should be upper case
- Loop conditions should be true at least once
- Loops should not contain more than a single "break" or "continue" statement
- Math operands should be cast before assignment
- Method names should comply with a naming convention
- Method parameters, caught exceptions and foreach variables should not be reassigned
- Methods "wait(...)", "notify()" and "notifyAll()" should never be called on Thread instances
- Methods named "equals" should override Object.equals(Object)
- Methods should not be empty
- Methods should not be named "hashcode" or "equal"
- Methods should not be too complex
- Methods should not have too many parameters
- Modifiers should be declared in the correct order
- Nested blocks of code should not be left empty
- Nested code blocks should not be used
- Non-constructor methods should not have the same name as the enclosing class
- Non-public methods should not be "@Transactional"
- Objects should not be created to be dropped immediately without being used
- Octal values should not be used
- Only static class initializers should be used
- Overriding methods should do more than simply call the same method in the super class
- Package declaration should match source file directory
- Package names should comply with a naming convention
- Parentheses should be removed from a single lambda input parameter when its type is inferred
- Primitive wrappers should not be instantiated only for "toString" or "compareTo" calls
- Printf-style format strings should not lead to unexpected behavior at runtime
- Public constants and fields initialized at declaration should be "static final" rather than merely "final"
- Public methods should throw at most one checked exception
- Redundant casts should not be used
- Reflection should not be used to check non-runtime annotations
- Related "if/else if" statements should not have the same condition
- Relational operators should be used in "for" loop termination conditions

- Return of boolean expressions should not be wrapped into an "if-then-else" statement
- Sections of code should not be "commented out"
- Servlets should never have mutable instance fields
- Short-circuit logic should be used in boolean contexts
- Source files should not have any duplicated blocks
- Standard outputs should not be used directly to log anything
- Statements should be on separate lines
- String literals should not be duplicated
- String.valueOf() should not be appended to a String
- Strings literals should be placed on the left side when checking for equality
- super.finalize() should be called at the end of Object.finalize() implementations
- Switch cases should end with an unconditional "break" statement
- Synchronization should not be based on Strings or boxed primitives
- Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used
- Tabulation characters should not be used
- The Array.equals(Object obj) method should not be used
- The default unnamed package should not be used
- The members of an interface declaration or class should appear in a pre-defined order
- The Object.finalize() method should not be called
- The Object.finalize() method should not be overriden
- The signature of "finalize()" should match that of "Object.finalize()"
- Thread.run() and Runnable.run() should not be called directly
- Throwable and Error should not be caught
- Throwable.printStackTrace(...) should not be called
- Throws declarations should not be superfluous
- Try-catch blocks should not be nested
- Type parameter names should comply with a naming convention
- Unused labels should be removed
- Unused local variables should be removed
- Unused method parameters should be removed
- Unused private fields should be removed
- Unused private method should be removed
- Useless "if(true) {...}" and "if(false){...}" blocks should be removed
- Useless imports should be removed
- Useless parentheses around expressions should be removed to prevent any misunderstanding
- Utility classes should not have public constructors
- Values passed to SQL commands should be sanitized

3.2. nDepend

A análise estática de código .NET é realizada pelo nDepend no qual avalia as métricas a seguir. A lista atualizada dessas métricas e mais informações podem ser obtidos em:

http://www.ndepend.com/docs/code-metrics

A lista foi obtida do site oficial, por isso está em inglês. Qualquer dúvida, o MCTQ está disponível para ajudar.

Metrics on application

NbLinesOfCode: (defined for application, assemblies, namespaces, types, methods) This metric (known as LOC) can be computed only if PDB files are present. NDepend computes this metric directly from the info provided in PDB files. The LOC for a method is equals to the number of sequence point found for this method in the PDB file. A sequence point is used to mark a spot in the IL code that corresponds to a specific location in the original source. More info about sequence points here.Notice that sequence points which correspond to C# braces'{' and '}' are not taken account.

Computing the number of lines of code from PDB's sequence points allows to obtain a logical LOC of code instead of a physical LOC (i.e directly computed from source files). 2 significant advantages of logical LOC over physical LOC are:

Coding style doesn't interfere with logical LOC. For example the LOC won't change because a method call is spawn on several lines because of a high number of argument.

logical LOC is independent from the language. Values obtained from assemblies written with different languages are comparable and can be summed.

Notice that the LOC for a type is the sum of its methods' LOC, the LOC for a namespace is the sum of its types' LOC, the LOC for an assembly is the sum of its namespaces' LOC and the LOC for an application is the sum of its assemblies LOC. Here are some observations:

Interfaces, abstract methods and enumerations have a LOC equals to 0. Only concrete code that is effectively executed is considered when computing LOC.

Namespaces, types, fields and methods declarations are not considered as line of code because they don't have corresponding sequence points.

When the C# or VB.NET compiler faces an inline instance fields initialization, it generates a sequence point for each of the instance constructor (the same remark applies for inline static fields initialization and static constructor).

LOC computed from an anonymous method doesn't interfere with the LOC of its outer declaring methods.

The overall ratio between NbILInstructions and LOC (in C# and VB.NET) is usually around 7.

Recommendations: Methods where NbLinesOfCode is higher than 20 are hard to understand and maintain. Methods where NbLinesOfCode is higher than 40 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool).

NbLinesOfComment: (defined for application, assemblies, namespaces, types, methods) (Only available for C# code, a VB.NET version is currently under development) This metric can be computed only if PDB files are present and if corresponding source files can be found. The number of lines of comment is computed as follow:

For a method, it is the number of lines of comment that can be found in its body. In C# the body of a method begins with a '{' and ends with a '}'. If a method contains an anonymous method, lines of comment defined in the anonymous method are not counted for the outer method but are counted for the anonymous method.

For a type, it is the sum of the number of lines of comment that can be found in each of its partial definition. In C#, each partial definition of a type begins with a '{ and ends with a '}'.

For a namespace, it is the sum of the number of lines of comment that can be found in each of its partial definition. In C# each partial definition of a namespace begins with a '{ and ends with a '}'.

For an assembly, it is the sum of the number of lines of comment that can be found in each of its source file.

Notice that this metric is not an additive metric (i.e for example, the number of lines of comment of a namespace can be greater than the number of lines of comment over all its types).

Recommendations: This metric is not helpful to asses the quality of source code. We prefer to use the metric PercentageComment.

PercentageComment: (defined for application, assemblies, namespaces, types, methods) (Only available for C# code, a VB.NET version is currently under development) This metric is computed with the following formula:

PercentageComment = 100*NbLinesOfComment / (NbLinesOfComment + NbLinesOfCode)

Recommendations: Code where the percentage of comment is lower than 20% should be more commented. However overly commented code (>40%) is not necessarily a blessing as it can be considered as an insult to the intelligence of the reader. Guidelines about code commenting can be found here.

NbILInstructions: (defined for application, assemblies, namespaces, types, methods) Notice that the number of IL instructions can vary depending if your assemblies are compiled in debug or in release mode. Indeed compiler's optimizations can modify the number of IL instructions. For example a compiler can add some nop IL instructions in debug mode to handle Edit and Continue and to allow attach an IL instruction to a curly brace. Notice that IL instructions of third-party assemblies are not taken account.

Recommendations: Methods where NbILInstructions is higher than 100 are hard to understand and maintain. Methods where NbILInstructions is higher than 200 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool).

NbAssemblies: (defined for application) Only application assemblies are taken into account.

NbNamespaces: (defined for application, assemblies) The number of namespaces. The anonymous namespace counts as one. If a namespace is defined over N assemblies, it will count as N. Namespaces declared in third-party assemblies are not taken account.

NbTypes: (defined for application, assemblies, namespaces) The number of types. A type can be an abstract or a concrete class, a structure, an enumeration, a delegate class or an interface. Types declared in third-party assemblies are not taken account.

NbMethods: (defined for application, assemblies, namespaces, types) The number of methods. A method can be an abstract, virtual or non-virtual method, a method declared in an interface, a constructor, a class constructor, a finalizer, a property/indexer getter or setter, an event adder or remover. Methods declared in third-party assemblies are not taken account.

Recommendations: Types where NbMethods > 20 might be hard to understand and maintain but there might be cases where it is relevant to have a high value for NbMethods. For example, the System.Windows.Forms.DataGridView third-party class has more than 1000 methods.

NbFields: (defined for application, assemblies, namespaces, types) The number of fields. A field can be a regular field, an enumeration's value or a readonly or a const field. Fields declared in third-party assemblies are not taken account.

Recommendations: Types that are not enumeration and where NbFields is higher 20 might be hard to understand and maintain but there might be cases where it is relevant to have a high value for NbFields. For example, the System.Windows.Forms.Control third-party class has more than 200 fields.

PercentageCoverage: (defined for application, assemblies, namespaces, types, methods) The percentage of code coverage by tests. Code coverage data are imported from coverage files. If you are using the uncoverable attribute feature on a method for example, if all sibling methods are 100% covered, then the parent type will be considered as 100% covered.

Coverage metrics are not available if the metric NbLinesOfCode is not available.

Recommendations: The closer to 100%, the better.

NbLinesOfCodeCovered: (defined for application, assemblies, namespaces, types, methods) The number of lines of code covered by tests.

NbLinesOfCodeNotCovered: (defined for application, assemblies, namespaces, types, methods) The number of lines of code not covered by tests.

Metrics on assemblies

By measuring coupling between types of your application, NDepend assesses the stability of each assembly. An assembly is considered stable if its types are used by a lot of types of third-party assemblies (i.e stable = painful to modify). If an assembly contains many abstract types (i.e interfaces and abstract classes) and few concrete types, it is considered as abstract. Thus, NDepend helps you detect which assemblies are potentially painful to maintain (i.e concrete and stable) and which assemblies are potentially useless (i.e abstract and instable).

Note: This theory and metrics have been first introduced by the excellent book Agile Software Development: Principles, Patterns, and Practices in C# Robert C. Martin (Prentice Hall PTR, 2006)

Afferent coupling (Ca): The number of types outside this assembly that depend on types within this assembly. High afferent coupling indicates that the concerned assemblies have many responsibilities.

Efferent coupling (Ce): The number of types outside this assembly used by child types of this assembly. High efferent coupling indicates that the concerned assembly is dependant. Notice that types declared in third-party assemblies are taken into account.

Relational Cohesion (H): Average number of internal relationships per type. Let R be the number of type relationships that are internal to this assembly (i.e that do not connect to types outside the assembly). Let N be the number of types within the assembly. H = (R + 1)/ N. The extra 1 in the formula prevents H=0 when N=1. The relational cohesion represents the relationship that this assembly has to all its types.

Recommendations: As classes inside an assembly should be strongly related, the cohesion should be high. On the other hand, too high values may indicate over-coupling. A good range for RelationalCohesion is 1.5 to 4.0. Assemblies where RelationalCohesion < 1.5 or RelationalCohesion > 4.0 might be problematic.

Instability (I): The ratio of efferent coupling (Ce) to total coupling. I = Ce / (Ce + Ca). This metric is an indicator of the assembly's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable assembly and I=1 indicating a completely instable assembly.

Abstractness (A): The ratio of the number of internal abstract types (i.e abstract classes and interfaces) to the number of internal types. The range for this metric is 0 to 1, with A=0 indicating a completely concrete assembly and A=1 indicating a completely abstract assembly.

Distance from main sequence (D): The perpendicular normalized distance of an assembly from the idealized line A + I = 1 (called main sequence). This metric is an indicator of the assembly's balance between abstractness and stability. An assembly squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal assemblies are either completely abstract and stable (I=0, A=1) or completely concrete and instable (I=1, A=0). The range for this metric is 0 to 1, with D=0 indicating an assembly that is coincident with the main sequence and D=1 indicating an assembly that is as far from the main sequence as possible. The picture in the report reveals if an assembly is in the zone of pain (I and A both close to 0) or in the zone of uselessness (I and A both close to 1).

Recommendations: Assemblies where NormDistFromMainSeq is higher than 0.7 might be problematic. However, in the real world it is very hard to avoid such assemblies.

Metrics on namespaces

Afferent coupling at namespace level (NamespaceCa): The Afferent Coupling for a particular namespace is the number of namespaces that depends directly on it.

Efferent coupling at namespace level (NamespaceCe): The Efferent Coupling for a particular namespace is the number of namespaces it directly depends on. Notice that namespaces declared in third-party assemblies are taken into account.

Level: (defined for assemblies, namespaces, types, methods) The Level value for a namespace is defined as follow:

Level = 0 : if the namespace doesn't use any other namespace.

Level = 1 : if the namespace only uses directly namespace defined in third-party assemblies.

Level = 1 + (Max Level over namespace it uses direcly)

Level = N/A : if the namespace is involved in a dependency cycle or uses directly or indirectly a namespace involved in a dependency cycle.

Level metric definitions for assemblies, types and methods are inferred from the above definition.

This metric has been first defined by John Lakos in his book Large-Scale C++ Software Design.

Recommendations: This metric helps objectively classify the assemblies, namespaces, types and methods as high level,mid level or low level. There is no particular recommendation for high or small values.

This metric is also useful to discover dependency cycles in your application. For instance if some namespaces are matched by the following CQLinq query, it means that there is some dependency cycles between the namespaces of your application:

from n in Application.Namespaces where n.Level == null select n

Metrics on types

Type rank: TypeRank values are computed by applying the Google PageRank algorithm on the graph of types' dependencies. A homothety of center 0.15 is applied to make it so that the average of TypeRank is 1.

Recommendations: Types with high TypeRank should be more carefully tested because bugs in such types will likely be more catastrophic.

Afferent Coupling at type level (Ca): The Afferent Coupling for a particular type is the number of types that depends directly on it.

Efferent Coupling at type level (Ce): The Efferent Coupling for a particular type is the number of types it directly depends on. Notice that types declared in third-party assemblies are taken into account.

Recommendations: Types where TypeCe > 50 are types that depends on too many other types. They are complex and have more than one responsibility. They are good candidate for refactoring.

Lack of Cohesion Of Methods (LCOM): The single responsibility principle states that a class should not have more than one reason to change. Such a class is said to be cohesive. A high LCOM value generally pinpoints a poorly cohesive class. There are several LCOM metrics. The LCOM takes its values in the range [0-1]. The LCOM HS (HS stands for Henderson-Sellers) takes its values in the range [0-2]. A LCOM HS value higher than 1 should be considered alarming. Here are algorithms used by NDepend to compute LCOM metrics:

LCOM = 1 – (sum(MF)/M*F)

LCOM HS = (M – sum(MF)/F)(M-1)

Where:

M is the number of methods in class (both static and instance methods are counted, it includes also constructors, properties getters/setters, events add/remove methods).

F is the number of instance fields in the class.

MF is the number of methods of the class accessing a particular instance field.

Sum(MF) is the sum of MF over all instance fields of the class.

The underlying idea behind these formulas can be stated as follow: a class is utterly cohesive if all its methods use all its instance fields, which means that sum(MF)=M*F and then LCOM = 0 and LCOMHS = 0.

Recommendations: Types where LCOM > 0.8 and NbFields > 10 and NbMethods >10 might be problematic. However, it is very hard to avoid such non-cohesive types. Types where LCOMHS > 1.0 and NbFields > 10 and NbMethods >10 should be avoided. Note that this constraint is stronger (and thus easier to satisfy) than the constraint types where LCOM > 0.8 and NbFields > 10 and NbMethods >10.

Cyclomatic Complexity (CC): (defined for types, methods) (Only available for C# code, a VB.NET version is currently under development) Cyclomatic complexity is a popular procedural software metric equal to the number of decisions that can be taken in a procedure. Concretely, in C# the CC of a method is 1 + {the number of following expressions found in the body of the method}:

if | while | for | foreach | case | default | continue | goto | && | || | catch | ternary operator ?: | ??

Following expressions are not counted for CC computation:

else | do | switch | try | using | throw | finally | return | object creation | method call | field access

The Cyclomatic Complexity metric is defined on methods. Adapted to the OO world, this metric is also defined for classes and structures as the sum of its methods CC. Notice that the CC of an anonymous method is not counted when computing the CC of its outer method.

Recommendations: Methods where CC is higher than 15 are hard to understand and maintain. Methods where CC is higher than 30 are extremely complex and should be split into smaller methods (except if they are automatically generated by a tool).

IL Cyclomatic Complexity (ILCC): The CC metric is language dependent. Thus, NDepend provides the ILCC which is language independent because it is computed from IL as 1 + {the number of different offsets targeted by a jump/branch IL instruction}. Experience shows that NDepend CC is a bit larger than the CC computed in C# or VB.NET. Indeed, a C# 'if' expression yields one IL jump. A C# 'for' loop yields two different offsets targeted by a branch IL instruction while a foreach C# loop yields three.

Recommendations: Methods where ILCyclomaticComplexity is higher than 20 are hard to understand and maintain. Methods where ILCyclomaticComplexity is higher than 40 are extremely complex and should be split into smaller methods (except if they are automatically generated by a tool).

Size of instance : (defined for instance fields and types) The size of instances of an instance field is defined as the size, in bytes, of instances of its type. The size of instance of a static field is equal to 0. The size of instances of a class or a structure is defined as the sum of size of instances of its fields plus the size of instances of its base class. Fields of reference types (class, interface, delegate…) always count for 4 bytes while the footprint of fields of value types (structure, int, byte, double…) might vary. Size of instances of an enumeration is equal to the size of instances of the underlying numeric primitive type. It is computed from the value__ instance field (all enumerations have such a field when compiled in IL). Size of instances of generic types might be erroneous because we can't statically know the footprint of parameter types (except when they have the class constraint).

Recommendations: Types where SizeOfInst is higher than 64 might degrade performance (depending on the number of instances created at runtime) and might be hard to maintain. However it is not a rule since sometime there is no alternative (the size of instances of the System.Net.NetworkInformation.SystemIcmpV6Statistics third-party class is 2064 bytes). Non-static and non-generic types where SizeOfInst is equal to 0 indicate stateless types that might eventually be turned into static classes.

NbInterfacesImplemented: The number of interfaces implemented. This metric is available for interfaces, in this case the value is the number of interface extended, directly or indirectly. For derived class, this metric also count the sum of interfaces implemented by base class(es).

Association Between Class (ABC): The Association Between Classes metric for a particular class or structure is the number of members of others types it directly uses in the body of its methods.

Number of Children (NOC): The number of children for a class is the number of sub-classes (whatever their positions in the sub branch of the inheritance tree). The number of children for an interface is the number of types that implement it. In both cases the computation of this metric only count types declared in the application code and thus, doesn't take account of types declared in third-party assemblies.

Depth of Inheritance Tree (DIT): The Depth of Inheritance Tree for a class or a structure is its number of base classes (including the System.Object class thus DIT >= 1).

Recommendations: Types where DepthOfInheritance is higher or equal than 6 might be hard to maintain. However it is not a rule since sometimes your classes might inherit from third-party classes which have a high value for depth of inheritance. For example, the average depth of inheritance for third-party classes which derive from System.Windows.Forms.Control is 5.3.

Metrics on methods

Method rank: MethodRank values are computed by applying the Google PageRank algorithm on the graph of methods' dependencies. A homothety of center 0.15 is applied to make it so that the average of MethodRank is 1.

Recommendations: Methods with high MethodRank should be more carefully tested because bugs in such methods will likely be more catastrophic.

Afferent coupling at method level (MethodCa): The Afferent Coupling for a particular method is the number of methods that depends directly on it.

Efferent coupling at method level (MethodCe): The Efferent Coupling for a particular method is the number of methods it directly depends on. Notice that methods declared in third-party assemblies are taken into account.

IL Nesting Depth: The metric Nesting Depth for a method is the maximum number of encapsulated scopes inside the body of the method. The metric IL Nesting Depth is computed from the IL code. Values computed are very similar to what we would expect by computing them from the C# or VB.NET source code.

When you have a testing condition with N conditions, such as if(i > 9 && i < 12) then it is considered as N scopes because it is possible to decompose such conditions into N atomic conditions.

When a method has a large number of case statements corresponding to a switch, the C# and VB.NET compiler generally produce optimizations while generating the IL. In such case, the IL Nesting Depth corresponding value might be slightly higher to what you would expect.

Recommendations: Methods where ILNestingDepth is higher than 4 are hard to understand and maintain. Methods where ILNestingDepth is higher than 8 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool).

NbParameters: The number of parameters of a method. Ref and Out are also counted. The this reference passed to instance methods in IL is not counted as a parameter.

Recommendations: Methods where NbParameters is higher than 5 might be painful to call and might degrade performance. You should prefer using additional properties/fields to the declaring type to handle numerous states. Another alternative is to provide a class or structure dedicated to handle arguments passing (for example see the class System.Diagnostics.ProcessStartInfo and the method System.Diagnostics.Process.Start(ProcessStartInfo)).

NbVariables: The number of variables declared in the body of a method.

Recommendations: Methods where NbVariables is higher than 8 are hard to understand and maintain. Methods where NbVariables is higher than 15 are extremely complex and should be split in smaller methods (except if they are automatically generated by a tool).

NbOverloads: The number of overloads of a method. . If a method is not overloaded, its NbOverloads value is equals to 1. This metric is also applicable to constructors.

Recommendations: Methods where NbOverloads is higher than 6 might be a problem to maintain and provoke higher coupling than necessary. This might also reveal a potential misused of the C# and VB.NET language that since C#3 and VB9 support object initialization. This feature helps reducing the number of constructors of a class.

PercentageBranchCoverage: (defined for methods) Branch coverage is a more accurate measure of coverage than PercentageCoverage because it compensates for method complexity. Since branch coverage is generated from the underlying opcodes, it often does not map cleanly to source code. That means it's difficult to take branch coverage values and determine how to write tests that will improve coverage.

Branch coverage is only available if your coverage data are imported from NCover™ coverage files and if the metric NbLinesOfCode is available.

Recommendations: The bottom line is:

Use PercentageCoverage as your measure of quality, but

Use PercentageBranchCoverage to determine which code statements need more testing.

It is also interesting to observe branch coverage values in conjunction with Code Source Cyclomatic Complexity values.

Metrics on fields

Afferent coupling at field level (FieldCa): The Afferent Coupling for a particular field is the number of methods that directly use it.