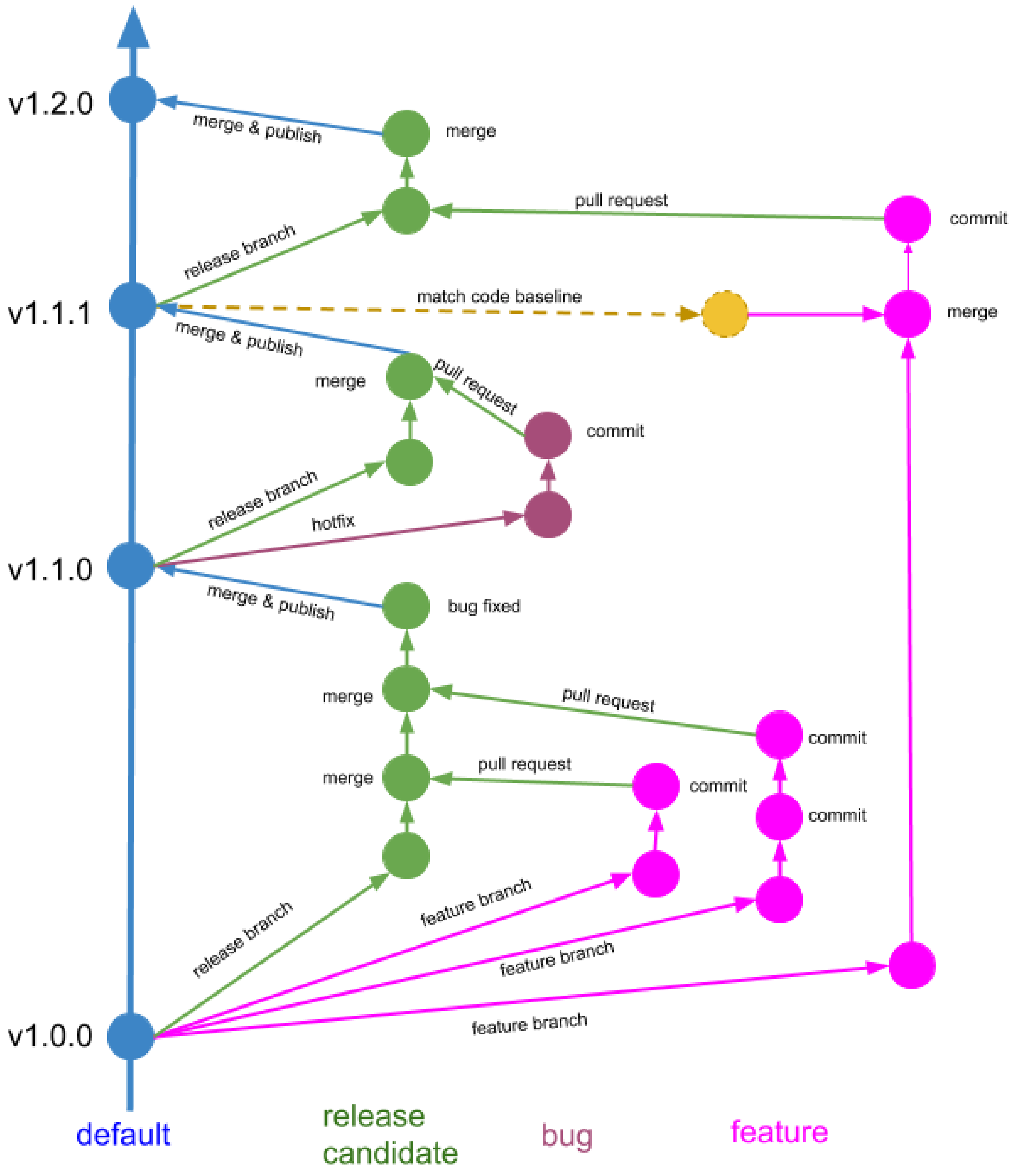
Applications Team - Coding Standards

Table of Contents

Revision Control

Branching



Versioning

Given a version number MAJOR.MINOR.PATCH, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards compatible manner, and
3. PATCH version when you make backwards compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the MAJOR.MINOR.PATCH format.

C# (.NET) Coding Standards

1. Use PascalCasing for class names and method names:

```
public class ClientActivity
{
  public void ClearStatistics()
  {
    //...
  }
  public void CalculateStatistics()
  {
    //...
  }
}
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

2. Use camelCasing for method arguments and local variables:

```
public class UserLog
{
  public void Add(LogEvent logEvent)
  {
    int itemCount = logEvent.Items.Count;
    // ...
  }
}
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

3. Do not use Hungarian notation or any other type identification in identifiers

```
// Correct
int counter;
string name;

// Avoid
int iCounter;
string strName;
```

Why: consistent with the Microsoft's .NET Framework and Visual Studio IDE makes determining types very easy (via tooltips). In general you want to avoid type indicators in any identifier.

4. Do not use Screaming Caps for constants or readonly variables:

```
// Correct
public const string ShippingType = "DropShip";

// Avoid
public const string SHIPPINGTYPE = "DropShip";
```

Why: consistent with the Microsoft's .NET Framework. Caps grab too much attention.

5. Use meaningful names for variables. The following example uses seattleCustomers for customers who are located in Seattle:

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

6. Avoid using Abbreviations. Exceptions: abbreviations commonly used as names, such as Id, Xml, Ftp, Uri.

```
// Correct
UserGroup userGroup;
Assignment employeeAssignment;

// Avoid
UserGroup usrGrp;
Assignment empAssignment;

// Exceptions
CustomerId customerId;
XmlDocument xmlDocument;
FtpHelper ftpHelper;
UriPart uriPart;
```

Why: consistent with the Microsoft's .NET Framework and prevents inconsistent abbreviations.

7. Use PascalCasing for abbreviations 3 characters or more (2 chars are both uppercase):

```
HtmlHelper htmlHelper;
FtpTransfer ftpTransfer;
UIControl uiControl;
```

Why: consistent with the Microsoft's .NET Framework. Caps would grab visually too much attention.

8. Do not use Underscores in identifiers. Exception: you can prefix private fields with an underscore:

```
// Correct
public DateTime clientAppointment;
public TimeSpan timeLeft;

// Avoid
public DateTime client_Appointment;
public TimeSpan time_Left;

// Exception (Class field)
private DateTime _registrationDate;
```

Why: consistent with the Microsoft's .NET Framework and makes code more natural to read (without 'slur'). Also avoids underline stress (inability to see underline).

9. Use predefined type names (C# aliases) like `int`, `float`, `string` for local, parameter and member declarations.
Use .NET Framework names like `Int32`, `Single`, `String` when accessing the type's static members like `Int32.TryParse` or `String.Join`.

```
// Correct
string firstName;
int lastIndex;
bool isSaved;
string commaSeparatedNames = String.Join(", ", names);
int index = Int32.Parse(input);
```

```
// Avoid
String firstName;
Int32 lastIndex;
Boolean isSaved;
string commaSeparatedNames = string.Join(", ", names);
int index = int.Parse(input);
```

Why: consistent with the Microsoft's .NET Framework and makes code more natural to read.

10. Use implicit type var for local variable declarations. Exception: primitive types (int, string, double, etc) use predefined names.

```
var stream = File.Create(path);
var customers = new Dictionary();

// Exceptions
int index = 100;
string timeSheet;
bool isCompleted;
```

Why: removes clutter, particularly with complex generic types. Type is easily detected with Visual Studio tooltips.

11. Do use noun or noun phrases to name a class.

```
public class Employee
{
}
public class BusinessLocation
{
}
public class DocumentCollection
{
}
```

Why: consistent with the Microsoft's .NET Framework and easy to remember.

12. Do prefix interfaces with the letter I. Interface names are noun (phrases) or adjectives.

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

Why: consistent with the Microsoft's .NET Framework.

13. Name source files according to their main classes. Exception: file names with partial classes reflect their source or purpose, e.g. designer, generated, etc.

```
// Located in Task.cs
public partial class Task
{
}

// Located in Task.generated.cs
public partial class Task
{
}
```

Why: consistent with the Microsoft practices. Files are alphabetically sorted and partial classes remain adjacent.

14. Organize namespaces with a clearly defined structure:

```
// Examples
namespace Company.Product.Module.SubModule
{
}
namespace Product.Module.Component
{
}
namespace Product.Layer.Module.Group
{
}
```

Why: consistent with the Microsoft's .NET Framework. Maintains good organization of your code base.

15. Vertically align curly brackets:

```
// Correct
class Program
{
  static void Main(string[] args)
  {
    //...
  }
}
```

Why: Microsoft has a different standard, but developers have overwhelmingly preferred vertically aligned brackets.

16. Declare all member variables at the top of a class, with static variables at the very top.

```
// Correct
public class Account
{
  public static string BankName;
  public static decimal Reserves;
  public string Number { get; set; }
  public DateTime DateOpened { get; set; }
  public DateTime DateClosed { get; set; }
  public decimal Balance { get; set; }
  // Constructor
  public Account()
  {
    // ...
  }
}
```

Why: generally accepted practice that prevents the need to hunt for variable declarations.

17. Use singular names for enums. Exception: bit field enums.

```
// Correct
public enum Color
{
  Red,
  Green,
  Blue,
  Yellow,
  Magenta,
  Cyan
}

// Exception
[Flags]
public enum Dockings
{
  None = 0,
  Top = 1,
  Right = 2,
  Bottom = 4,
  Left = 8
}
```

Why: consistent with the Microsoft's .NET Framework and makes the code more natural to read. Plural flags because enum can hold multiple values (using bitwise 'OR').

18. Do not explicitly specify a type of an enum or values of enums (except bit fields):

```
// Don't
public enum Direction : long
{
  North = 1,
  East = 2,
  South = 3,
  West = 4
}

// Correct
public enum Direction
{
  North,
  East,
  South,
  West
}
```

Why: can create confusion when relying on actual types and values.

19. Do not use an "Enum" suffix in enum type names:

```
// Don't
public enum CoinEnum
{
  Penny,
  Nickel,
  Dime,
  Quarter,
  Dollar
}

// Correct
public enum Coin
{
  Penny,
  Nickel,
  Dime,
  Quarter,
  Dollar
}
```

Why: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

20. Do not use "Flag" or "Flags" suffixes in enum type names:

```
// Don't
[Flags]
public enum DockingsFlags
{
  None = 0,
  Top = 1,
  Right = 2,
  Bottom = 4,
  Left = 8
}

// Correct
[Flags]
public enum Dockings
{
  None = 0,
  Top = 1,
  Right = 2,
  Bottom = 4,
  Left = 8
}
```

Why: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

21. Use suffix EventArgs at creation of the new classes comprising the information on event:

```
// Correct
public class BarcodeReadEventArgs : System.EventArgs
{
}
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

22. Name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

```
public delegate void ReadBarcodeEventHandler(object sender,
ReadBarcodeEventArgs e);
```

Why: consistent with the Microsoft's .NET Framework and easy to read.

23. Do not create names of parameters in methods (or constructors) which differ only by the register:

```
// Avoid
private void MyFunction(string name, string Name)
{
  //...
}
```

Why: consistent with the Microsoft's .NET Framework and easy to read, and also excludes possibility of occurrence of conflict situations.

24. Use two parameters named sender and e in event handlers. The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

```
public void ReadBarcodeEventHandler(object sender, ReadBarcodeEventArgs e)
{
  //...
}
```

Why: consistent with the Microsoft's .NET Framework and consistent with prior rule of no type indicators in identifiers.

25. Do use suffix Exception at creation of the new classes comprising the information on exception:

```
// Correct
public class BarcodeReadException : System.Exception
{
}
```

Why: consistent with the Microsoft's .NET Framework and easy to read.