Order of class members

```
// Public static readonly
// Private static readonly
// Private static
// Fields set by Unity
    [SerializeField]
    private WorkshopSpecification _workshopSpecification;
// Events
    public event EventHandler<StormStartedEventArgs> StormStarted;
// Auto-properties
    public int DesiredWorkers { get; private set; }
// Dependencies
    private EventBus _eventBus;
// Private fields - components retrieved in Awake
    private Inventory _inventory;
// Private fields - others
    private float _progress;
// Constructors
    public MyClass(...)
// Static factory methods
    public static MyClass Create(...)
// InjectDependencies
    public void InjectDependencies(...)
// Awake
// Start
// OnDestroy, OnEnable, OnDisable etc.
// FixedUpdate
// Update
// Public properties
// Public instance methods
// Public static methods
// Private properties
    private int DesiredWorkers {
      get { return _desiredWorkers; }
      set { _desiredWorkers = value; }
    }
// Private methods
// Private classes, structs, records
```

Naming

- Use names that don't reflect used type
  - i.e. `description` when using StringBuilder (instead of `stringBuilder`)
  - i.e. `items` when using List<T> (instead of `list`)

  Names of Classes, Structs, and Interfaces - Framework Design Guidelines

  C# identifier names | Microsoft Docs

Types

- Types that can be internal should be internal
- Prefer using types and methods defined in .Net over Unity. For example use System.Math instead of UnityEngine.Mathf if possible.
- When implementing interface members, they should be kept in the same order as in the interface and if possible grouped together.
- Add the "Attribute" suffix to attributes

```
public MyAttribute : Attribute {}
```

Events

- Do not derive from EventArgs.
- Use a struct instead of a class when needed due to performance reasons. Remember that in some cases a class can be a more performant choice.
- Declaring class:

```
public event EventHandler<StormStartedEventArgs> StormStarted;
```

- Argument type:

```
public class StormStartedEventArgs {
  public int NumberOfLightnings { get; }
  public StormStartedEventArgs(int numberOfLightnings) {
    NumberOfLightnings = numberOfLightnings;
  }
}
```

- Listening class:
  - Using a method:

```
StormStarted += OnStormStarted;
private void OnStormStarted(object sender,
                            StormStartedEventArgs e) {
  ...
```

```
}
```

- Using a lambda:

```
StormStarted += (_, _) => DoStuff();
```

Blank lines

- Methods must be surrounded by single blank lines.
- A single blank line may be used between fields, events and properties to create logical groupings if it improves readability.
- A single blank line may be used inside methods if it improves readability.
- Multiple consecutive blank lines are not permitted.

Type members

- Private methods than can be static should be static
- Use readonly for fields whenever possible
- Use `public/private static readonly` instead of `public/private const`
- Don't initialize fields and properties to their default value
- Private methods should be arranged in the order of use. Calling method should be above the called one.
- Don't use the expression body definition (`member => expression;`) for methods. Always use a full statement body.

Methods

- Always use base types for parameters, except when:
  - You want to make sure that the caller will only be able to pass a specific type of component.

```
private void RemoveYielder(TreeComponent treeComponent) {
  var blockObject = treeComponent.GetComponent<BlockObject>();
  _yieldersInArea.Remove(blockObject.Coordinates);
}
```

- Method parameters should be in as few lines as possible, except when:
  - It might impair readability or mislead - for example a method with one, then two and then again one parameter in a line (due to their length).
  - When injecting dependencies (Singleton constructors / InjectDependencies method) - then each parameter should be in its own line.
- Use `var` for variables whenever possible.
- Local functions are allowed in two cases:

    - To simplify implementation of recursive algorithms.
    - To avoid implicitly captured closures when using lambda expressions.

Constructors / Inject dependencies

- When injecting dependencies (Singleton constructors / InjectDependencies method)
    - Each parameter should be in its own line.
    - Field declarations, method parameters and in method assignment should all be in the same order.
- When constructing types:
    - Constructors should not contain any logic, only field assignment. It is permitted to use methods like ToList or ToImmutableArray to bring parameter type to field type. If additional logic is needed, it should be performed elsewhere (for example in Load/Initialize methods or in factory)
    - Constructor parameters order:
        - Injected dependencies (for example ILoc or Tooltip) should be first parameters followed by remaining ones.
        - In the constructor, fields and properties should be assigned in the same order as declared parameters. Preferably, this will also be the order of fields and properties in the owning type.

UI Toolkit

- Use VisualElement.RegisterCallback<ClickEvent> to assign callback action to the visual element.
    - Using “Button.clicked += Action” or other click events (like MouseDownEvent) to simulate click action is prohibited, as they interfere with the ClickEvent callbacks.
    - Using additional events besides ClickEvent is allowed in special cases, but even in these cases, ClickEvent should be used to process clicks.

Refactoring

- When performing:
    - Assembly, file, type or type member rename
    - File move between assemblies
    - Formatting changes
    - Other code readability changes which do not affect the behavior of the code

  don’t combine it with any other code change. It makes testing easier.

Code review

- When testing an issue, first perform functional checks and wait for functional fixes, if needed. When the functional aspect of the issue is working as expected, perform code review.
- If the issue has no problems, close it. Otherwise provide feedback and move it back to in-progress and re-assign the person responsible for the issue.
- When your issue is being tested, wait with fixes until you are reassigned and the issue is moved back to in-progress.
- When all feedback is resolved, move the issue to test and re-assign the person responsible for testing.

Backward compatibility

- When using GetSingleton/GetComponent in Save/Load methods always store returned value in a variable with the name same as the class name.
  class: `WaterMap`
  ```
  var waterMap = _singletonLoader.GetSingleton(WaterMapKey);
  ```
- Persistence keys should reflect the name of stored items.
  ```
  psr PropertyKey<Dwelling> HomeKey = new("Home");
  public Dwelling Home { get; private set; }
  ```
- ILoadableSingleton should always account for data not present in SingletonLoader in method Load and initialize self accordingly if needed
  ```
  public void Load() {
    if (_singletonLoader.HasSingleton(PlantingServiceKey)) {
      [...]
      _plantingMap = plantingService.Get(PlantingMapKey);
    } else {
      _plantingMap = new PlantingMap(_terrainService.Size.XY());
    }
  }
  ```
- When modifying persistent singleton or entity, changes should be reflected in keys and Load method. If the old data is not needed, backward compatibility checks can be omitted.
  - If a member was added perform a check if it is present in loader
  - If class or any of persisted members was renamed, check under both new key name (default check) and under old key name (fallback check if first check failed).
  - If a member was moved to another component fallback check should use the origin component name.

- Load method with backward compatibility code should be decorated with the BackwardCompatible attribute with the date of modification. When new backward compatibility code is added to an already decorated method, the date should be updated.
- Backward compatibility keys should be created inside the Load method.

Should read

- Clean Code: A Handbook of Agile Software Craftsmanship - Robert C. Martin
- Effective C#: 50 Specific Ways to Improve Your C# - Bill Wagner