# 3902 Code Reviews

Your Name: Tony Mai

Who’s code are you reviewing: Nick, Ben (ConcreteSprite.cs)

#### 1. Code formatting

While going through the code, check the code formatting to improve readability and ensure that there are no blockers:

1. Use alignments (left margin), proper white space. Also ensure that code block starting point and ending point are easily identifiable.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Ensure that proper naming conventions (Pascal, CamelCase etc.) have been followed.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Code should fit in the standard 14 inch laptop screen.  There shouldn’t be a need to scroll horizontally to view the code. In a 21 inch monitor, other windows (toolbox, properties etc.) can be opened while modifying code, so always write code keeping in view a 14 inch monitor.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Remove the commented code

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

#### 2. Coding best practices

1. No hard coding, use constants/configuration values.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Group similar values under an [enumeration](https://en.wikipedia.org/wiki/Enumerated_type) (enum).

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Good and comprehensive commenting

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Avoid multiple if/else blocks.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

#### 3. Non Functional requirements

**a) Maintainability (Supportability)** – The application should require the least amount of effort to support in near future. It should be easy to identify and fix a defect.

1. **Readability:** Code should be self-explanatory. *Get a feel of story reading, while going through the code*. Use appropriate name for variables, functions and classes. If you are taking more time to understand the code, then either code needs refactoring or at least comments have to be written to make it clear.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

*The name of the class is strange. There appears to be a misunderstanding as to what a “sprite” is.*

1. **Testability:** The code should be easy to test. Refactor into a separate function (if required). Use interfaces while talking to other layers, as interfaces can be mocked easily. Try to avoid static functions, singleton classes as these are not easily testable by mocks.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

**b) Reusability**

1. DRY (Do not Repeat Yourself) principle: The same code should not be repeated more than twice.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. **Extensibility –**Easy to add enhancements with minimal changes to the existing code. One component should be easily replaceable by a better component.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

#### 4. Object-Oriented Analysis and Design (OOAD) Principles

1. **Single Responsibility Principle (SRS):** Do not place more than one responsibility into a single class or function, refactor into separate classes and functions.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. **Open Closed Principle:** While adding new functionality, existing code should not be modified. New functionality should be written in new classes and functions.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. [**Liskov substitutability principle**](https://en.wikipedia.org/wiki/Liskov_substitution_principle)**:** The child class should not change the behavior (meaning) of the parent class. The child class can be used as a substitute for a base class.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. **Interface segregation:** Do not create lengthy interfaces, instead split them into smaller interfaces based on the functionality. The interface should not contain any dependencies (parameters), which are not required for the expected functionality.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. **Dependency Injection:** Do not hardcode the dependencies, instead inject them.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

1. Low Coupling and high Cohesion.

| Yes | Somewhat | No | N/A |
| --- | --- | --- | --- |

*This class seems to be the main Entity class that represents a character game object. It currently lacks all of the methods such as move, takeDamage, etc. that character entities should have. These methods instead lie elsewhere in other classes.*

*High coupling due to other classes needing to know a lot about a ConcreteSprite object, since ConcreteSprite lacks essential methods to alter itself.*

*Low cohesion due to ConcreteSprite’s methods being scattered across multiple classes.*