# Technische Beschreibung

Die Initialize-Funktion wird in Game1.cs ausgeführt und setzt direkt den aktuellen State, welcher für die folgenden LoadContent(), Update() und Draw() Funktionen verwendet wird.

```
protected override void Initialize()
{
    _graphics.PreferredBackBufferWidth = 1920;  // set this value to the desired width of your window
    _graphics.PreferredBackBufferHeight = 1080;   // set this value to the desired height of your window
    _graphics.IsFullScreen = true;

    _graphics.ApplyChanges();

    ScreenWidth = _graphics.PreferredBackBufferWidth;
    ScreenHeight = _graphics.PreferredBackBufferHeight;
    _currentState = new IntroState(this, _graphics, Content);


    base.Initialize();
}
```

Die Game1.cs ist nicht selbst für das Laden des Contents verantwortlich, sondern übergibt die Aufgabe an den jeweiligen State.

```
protected override void LoadContent()
{
    _spriteBatch = new SpriteBatch(GraphicsDevice);
    _currentState.LoadContent();

}
```

Der State wird mit der public Funktion ChangeState() geändert und ruft selbst die LoadContent() Funktion auf, um die jeweiligen Contents des neuen States zu laden.

```
public void ChangeState(State state)
{
    _currentState = state;

    _currentState.LoadContent();
}
```

Die Game1.cs ist nicht selbst für die Update-Funktion verantwortlich, sondern übergibt die Aufgabe an den jeweiligen State.

```
protected override void Update(GameTime gameTime)
{

    _currentState.Update(gameTime);

    base.Update(gameTime);
}
```

Die Game1.cs ist nicht selbst für die Draw-Funktion verantwortlich, sondern übergibt die Aufgabe an den jeweiligen State.

```
protected override void Draw(GameTime gameTime)
{
    _spriteBatch.Begin();

    _currentState.Draw(gameTime, _spriteBatch);

    _spriteBatch.End();

    base.Draw(gameTime);
}
```

Die jeweiligen States haben eine Liste an IGameParts, welches als Interfaces definiert wurde.

```
public interface IGameParts
{
    19 references
    void Update(GameTime gameTime, List<IGameParts> gameParts, List<Tower> backgroundTowers = null);

    21 references
    void Draw(GameTime gameTime, SpriteBatch spriteBatch);

}
```

Jede Klasse, auf die Update() und Draw() aufgerufen werden muss, implementiert das IGameParts Interface. Hier am Beispiel der Klasse TowerButton

```
namespace Tower_Defence.Buttons
{
    8 references
    public class TowerButton : Button, IGameParts
    {
```

Jeder State definiert seine benötigten Objekte selbst oder über eine dafür vorgesehene Klasse, wie den EnemySpawner, der die Enemy-Objekte erstellt.

Diese Instanzen werden dann im jeweiligen State der Liste an IGameParts hinzugefügt.

Für jedes IGamePart wird in der Folge Update() und Draw() aufgerufen, welches durch das Interface implementiert wurde.

```
foreach (IGameParts gamePart in _gameParts.ToArray())
{
    if (gamePart is null) { continue; }
    gamePart.Update(gameTime, _gameParts, _backgroundTowers);
}
```

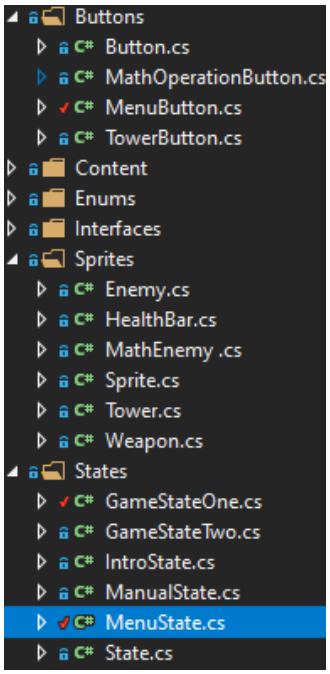
```
foreach (IGameParts gamePart in _gameParts)
{
    if (gamePart is MenuButton)
    {
        gamePart.Draw(gameTime, spritebatch);
    }
}
```

Jedes Objekt ist dadurch für sein eigenes Update und Draw selbst zuständig. Hier am Beispiel der Enemy.cs.

```
public override void Update(GameTime gameTime, List<IGameParts> gameParts, List<Tower> backgroundTowers)
{
    if(!idle)
    {
        EnemyMovement(gameTime);
    }
    _animationManager.Update(gameTime);

}

21 references
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    _animationManager.Draw(spriteBatch);

}
```

Die für das Game benötigten Objektinstanzen erben jeweils von einer abstrakten Base Class.

Base Classes: Button, Sprite, State



Um im jeweiligen State auf relevante Änderungen der einzelnen Instanzen reagieren zu können (z.B. Button-Klick, Enemie-Death) wurden Events verwendet, an die sich die Instanzen anmelden.

Klasse TowerButton

```
public class TowerButton : Button, IGameParts
{
    public bool isPreviewOn;

    public event Action<AttackType> PreviewTowerEventHandler;
    public event Action TurnTowerPreviewOffEventHandler;
    public event Action<TowerButton> PlaceTowerEventHandler;
```

Instanz der Klasse TowerButton im GameState

```
archerButton = new TowerButton(_archerTowerButton, AttackType.archer)
{
    Position = new Vector2(50, 30),
    Scale = 0.7f
};

archerButton.PreviewTowerEventHandler += HandleTowerPreview;
archerButton.PlaceTowerEventHandler += HandlePlaceTower;
archerButton.TurnTowerPreviewOffEventHandler += HandleTurnPreviewOff;
```

Zusätzlich wurde für allgemein benötigte Variablen mit einem GameManager experimentiert, der als Singleton implementiert ist.

```
public sealed class GameManager
{

    56 references
    public static GameManager GameManagerInstance { get { return Nested.gameManagerInstance; } }

    13 references
    public Difficulty Difficulty { get; set; }
    8 references
    public int StoppedEnemies { get; set; }
    8 references
    public bool MathOperationIsUsed { get; set; }
    5 references
    public bool EnemyIsHit { get; set; }
    22 references
    public Level CurrentLevel { get; set; }
    1 reference
    private GameManager()
    {

    }

    2 references
    private class Nested
    {
        0 references
        static Nested()
        {

        }
        internal static readonly GameManager gameManagerInstance = new GameManager();
    }
```

Für AttackType, Difficulty, Level und MathOperation wurden Enums definiert, um Code zu reduzieren. Beispielsweise wird beim Auswählen eines Towers über einen TowerButton geprüft, welcher AttackType der jeweilige Tower hat und darüber in einem Dictionnary die jeweilige Sprite gefunden, um den Tower zu zeichnen.