

[illegible]

<code>&lt;&lt;structD&gt;&gt;</code> <b>MinMaxGeneric</b>	<code>&lt;&lt;structD&gt;&gt;</code> <b>Size2DGeneric</b>
+ Min : Generic <<property>> + Max : Generic <<property>>	+ Width : Generic <<property>> + Height : Generic <<property>>
+ MinMax(Generic min, Generic max)	+ MinMax(Generic w, Generic h)

```

classDiagram
    class Factory {
        <<abstract>>
        +age: float
        +background: starColors
        +age: double
        +useStar: bool
    }
    class Stars {
        +color: List<Color>
        +colorName: int
        +size: float
        +size: int
        +background: int
        +img: Random
        +texture: Texture2D
        +diffuse: Material
        +rect: Rect
        +diffAngle: float
        +colorValue: Color
    }
    class Background {
        +stars: List<Star>
        +bgColor: Color
    }
    class MenuScreenData {
        +mouse: Vector2
        +animation: Dictionary<ScreenType, string>
        +MenuScreenData(BrowserMenuData screen)
        +AddScreenType(screenType: string)
        +RemoveScreenType(screenType: string)
    }
    Factory <|-- Factory
    Factory --> Stars
    Factory --> Background
    Stars --> Background
    Stars --> MenuScreenData
    Background --> MenuScreenData
  
```

The diagram illustrates the architecture of the Star Wars application. It features four main classes: **Factory** (an abstract class), **Stars**, **Background**, and **MenuScreenData**.

- Factory** (Abstract Class):
  - Attributes: `age: float`, `background: starColors`, `age: double`, `useStar: bool`.
  - Relationships: Generalized by `Factory` (concrete class). Associated with `Stars` and `Background`.
- Stars**:
  - Attributes: `color: List<Color>`, `colorName: int`, `size: float`, `size: int`, `background: int`, `img: Random`, `texture: Texture2D`, `diffuse: Material`, `rect: Rect`, `diffAngle: float`, `colorValue: Color`.
  - Relationships: Associated with `Background` and `MenuScreenData`.
- Background**:
  - Attributes: `stars: List<Star>`, `bgColor: Color`.
  - Relationships: Associated with `MenuScreenData`.
- MenuScreenData**:
  - Attributes: `mouse: Vector2`, `animation: Dictionary<ScreenType, string>`.
  - Operations: `MenuScreenData(BrowserMenuData screen)`, `AddScreenType(screenType: string)`, `RemoveScreenType(screenType: string)`.

The diagram shows a clear separation of concerns, with the **Factory** class managing the creation and state of the **Stars** and **Background** objects, which are then rendered and managed by the **MenuScreenData** class.

# Game

```
classDiagram
    class Team {
        +Team1
        +Team2
        +Team3
        +Player
        +Computer
    }
    class PlayerShip {
        +Ship
    }
    class State {
        +COOLDOWN
        +READY
    }
    Team "1" -- "1" PlayerShip
    PlayerShip "1" -- "1" State
```

The diagram illustrates the relationships between three classes: Team, PlayerShip, and State. The Team class contains attributes Team1, Team2, Team3, Player, and Computer. The PlayerShip class contains the attribute Ship. The State class contains attributes COOLDOWN and READY. There is a directed association from Team to PlayerShip, and another directed association from PlayerShip to State.

```

classDiagram
    class Game {
        +gameType: SelectionMenu or HighScoreMenu
        +play() void
    }
    class SelectionMenu {
        +select() void
        +selectGroup() SelectionGroup
    }
    class HighScoreMenu {
        +highScores: List<HighScore>
        +display() void
    }
    class SelectionGroup {
        +groupType: SelectionMenu or HighScoreMenu
        +select() void
    }
    Game <|-- SelectionMenu
    Game <|-- HighScoreMenu
    SelectionMenu --> SelectionGroup
    SelectionMenu --> HighScoreMenu
    SelectionGroup --> SelectionMenu
    SelectionGroup --> HighScoreMenu
  
```

The diagram illustrates the structure of the 'Game' system. It features a base class 'Game' and two subclasses: 'SelectionMenu' and 'HighScoreMenu'. The 'Game' class has a 'gameType' attribute (which can be either 'SelectionMenu' or 'HighScoreMenu') and a 'play()' method. The 'SelectionMenu' class has a 'select()' method and a 'selectGroup()' method that returns a 'SelectionGroup' object. The 'HighScoreMenu' class has a 'display()' method and a 'highScores' attribute (a list of 'HighScore' objects). The 'SelectionGroup' class has a 'groupType' attribute (which can be either 'SelectionMenu' or 'HighScoreMenu') and a 'select()' method. The diagram also shows a 'Game' class with a 'game' attribute and a 'play()' method. The diagram is a simplified version of the one in the text, focusing on the core classes and their relationships.

# Menu Elements

```

classDiagram
    class Difficulty {
        int id
        String name
        int spawnTime
        int add
        int diffLevel
        boolean locked
    }
    class DifficultyManager {
        spawnTime()
        add()
        diffLevel()
        locked()
    }
    DifficultyManager --> Difficulty
  
```

<pre>         struct Command {         private:             virtual ~Command() {}         public:             virtual void execute() const = 0;         };          struct ShootCommand : Command {         private:             int x;             int y;         public:             ShootCommand(int x, int y) : x(x), y(y) {}             void execute() const {                 // ...             }         };     </pre>
---

[illegible]

<pre>&lt;&lt;abstract&gt;&gt; ComponentFactory</pre>
--

```

classDiagram
    class Shape {
        -shape: List<List<Segment>>
        -length: float
        -pos: Point2D
        +Shape(List<List<Segment>>, float length, Point2D pos)
        +move(Point2D target)
        +move(Point2D target, float speed)
        +move(Point2D target, float speed, float time)
        +getLength() List<List<Segment>>
    }
    class ShapeFactory {
        -shape: List<List<Segment>>
        -length: float
        -pos: Point2D
        +BoundingBox() List<List<Segment>> +consolidate() property
        +Make() List<List<Segment>>
        +CreateObject(shapeCds, float scale, Point2D pos)
        +AddSegment(List<List<Segment>>, float length)
        +AddSegment(List<List<Segment>>, float length, float speed, float time)
        +BoxContains(Point2D pos, float length, Point2D)
        +CreateBoundingBox(List<List<Segment>>, List<List<Segment>>)
    }
    class Detail_Mover {
        -boxen: float
        -velocity: float
        -Coordinate: List<List<Segment>> +consolidate()
        -shaper: CoordinateShaper
        +Detective() float pos, shape, colors, health, wt, dv,
        +move, velocity, distance, boundaryStrategy, time)
        +update()
        +GetTime() float
    }
    class Detail_Factory {
        +CreateListSegment(), Point2D pos, Detail
    }
    ShapeFactory --> Shape
    ShapeFactory --> Detail_Factory
    Detail_Mover --> Shape
    Detail_Mover --> Detail_Factory

```

