

PEMROGRAMAN LANJUT

Large Class and God Class

Oleh

Tri Hadiyah Muliawati

Politeknik Elektronika Negeri Surabaya

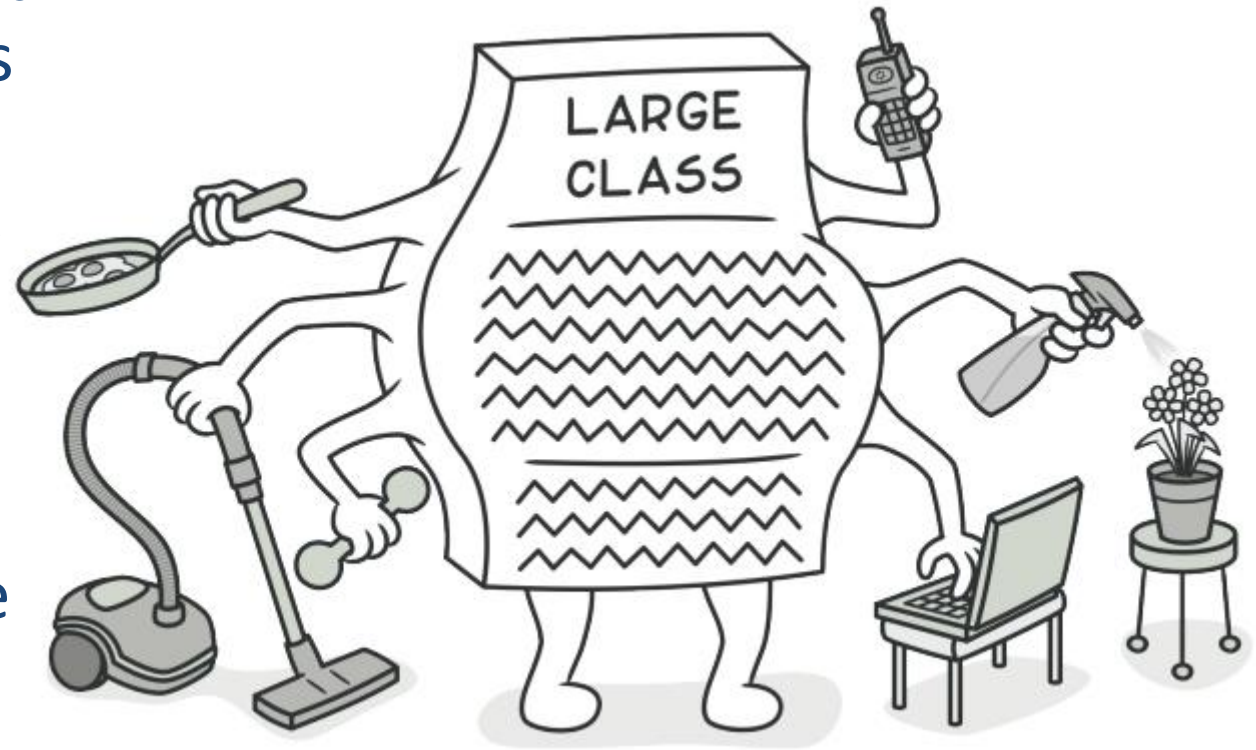
2021



Politeknik Elektronika Negeri Surabaya
Departemen Teknik Informatika dan Komputer

Large Class

- Classes usually start small. But over time, they get bloated as the program grows.
- It belongs to bloater category in code smell.
- Large class is not defined by length of LoC (Line of Code), but its violation of SRP (Single Responsibility Principle).



Large Class: Refactoring

- **Extract Class:** if part of the behavior of the large class can be spun off into a separate component.
- **Extract Subclass:** if part of the behavior of the large class can be implemented in different ways or is used in rare cases.
- **Extract Interface:** if it's necessary to have a list of the operations and behaviors that the client can use.
- If a large class is responsible for the graphical interface, you may try to move some of its data and behavior to a separate domain object.



```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace Trivia
6  {
7      public class Game
8      {
9          private readonly List<string> _players = new List<string>();
10
11         private readonly int[] _places = new int[6];
12         private readonly int[] _purses = new int[6];
13
14         private readonly bool[] _inPenaltyBox = new bool[6];
15
16         private readonly LinkedList<string> _popQuestions = new LinkedList<string>();
17         private readonly LinkedList<string> _scienceQuestions = new LinkedList<string>();
18         private readonly LinkedList<string> _sportsQuestions = new LinkedList<string>();
19         private readonly LinkedList<string> _rockQuestions = new LinkedList<string>();
20
21         private int _currentPlayer;
22         private bool _isGettingOutOfPenaltyBox;
23
24         public Game()
25         {
26             for (var i = 0; i < 50; i++)
27             {
28                 _popQuestions.AddLast("Pop Question " + i);
29                 _scienceQuestions.AddLast("Science Question " + i);
30                 _sportsQuestions.AddLast("Sports Question " + i);
31                 _rockQuestions.AddLast(CreateRockQuestion(i));
32             }
33         }
34     }

```

Sumber: <https://github.com/emilybache/trivia>

```
35     public string CreateRockQuestion(int index)
36     {
37         return "Rock Question " + index;
38     }
39
40     public bool IsPlayable()
41     {
42         return (HowManyPlayers() >= 2);
43     }
44
45     public bool Add(string playerName)
46     {
47         _players.Add(playerName);
48         _places[HowManyPlayers()] = 0;
49         _purses[HowManyPlayers()] = 0;
50         _inPenaltyBox[HowManyPlayers()] = false;
51
52         Console.WriteLine(playerName + " was added");
53         Console.WriteLine("They are player number " + _players.Count);
54         return true;
55     }
56
57     public int HowManyPlayers()
58     {
59         return _players.Count;
60     }
61
```

```

62 public void Roll(int roll)
63 {
64     Console.WriteLine(_players[_currentPlayer] + " is the current player");
65     Console.WriteLine("They have rolled a " + roll);
66
67     if (_inPenaltyBox[_currentPlayer])
68     {
69         if (roll % 2 != 0)
70         {
71             _isGettingOutOfPenaltyBox = true;
72
73             Console.WriteLine(_players[_currentPlayer] + " is getting out of the penalty box");
74             _places[_currentPlayer] = _places[_currentPlayer] + roll;
75             if (_places[_currentPlayer] > 11) _places[_currentPlayer] = _places[_currentPlayer] - 12;
76
77             Console.WriteLine(_players[_currentPlayer]
78                 + "'s new location is "
79                 + _places[_currentPlayer]);
80             Console.WriteLine("The category is " + CurrentCategory());
81             AskQuestion();
82         }
83         else
84         {
85             Console.WriteLine(_players[_currentPlayer] + " is not getting out of the penalty box");
86             _isGettingOutOfPenaltyBox = false;
87         }
88     }
89     else
90     {
91         _places[_currentPlayer] = _places[_currentPlayer] + roll;
92         if (_places[_currentPlayer] > 11) _places[_currentPlayer] = _places[_currentPlayer] - 12;
93
94         Console.WriteLine(_players[_currentPlayer]
95             + "'s new location is "
96             + _places[_currentPlayer]);
97         Console.WriteLine("The category is " + CurrentCategory());
98         AskQuestion();
99     }
100 }
101

```

Sumber: <https://github.com/emilybache/trivia>

```
102 private void AskQuestion()
103 {
104     if (CurrentCategory() == "Pop")
105     {
106         Console.WriteLine(_popQuestions.First());
107         _popQuestions.RemoveFirst();
108     }
109     if (CurrentCategory() == "Science")
110     {
111         Console.WriteLine(_scienceQuestions.First());
112         _scienceQuestions.RemoveFirst();
113     }
114     if (CurrentCategory() == "Sports")
115     {
116         Console.WriteLine(_sportsQuestions.First());
117         _sportsQuestions.RemoveFirst();
118     }
119     if (CurrentCategory() == "Rock")
120     {
121         Console.WriteLine(_rockQuestions.First());
122         _rockQuestions.RemoveFirst();
123     }
124 }
125
126 private string CurrentCategory()
127 {
128     if (_places[_currentPlayer] == 0) return "Pop";
129     if (_places[_currentPlayer] == 4) return "Pop";
130     if (_places[_currentPlayer] == 8) return "Pop";
131     if (_places[_currentPlayer] == 1) return "Science";
132     if (_places[_currentPlayer] == 5) return "Science";
133     if (_places[_currentPlayer] == 9) return "Science";
134     if (_places[_currentPlayer] == 2) return "Sports";
135     if (_places[_currentPlayer] == 6) return "Sports";
136     if (_places[_currentPlayer] == 10) return "Sports";
137     return "Rock";
138 }
139
```

Sumber: <https://github.com/emilybache/trivia>

```

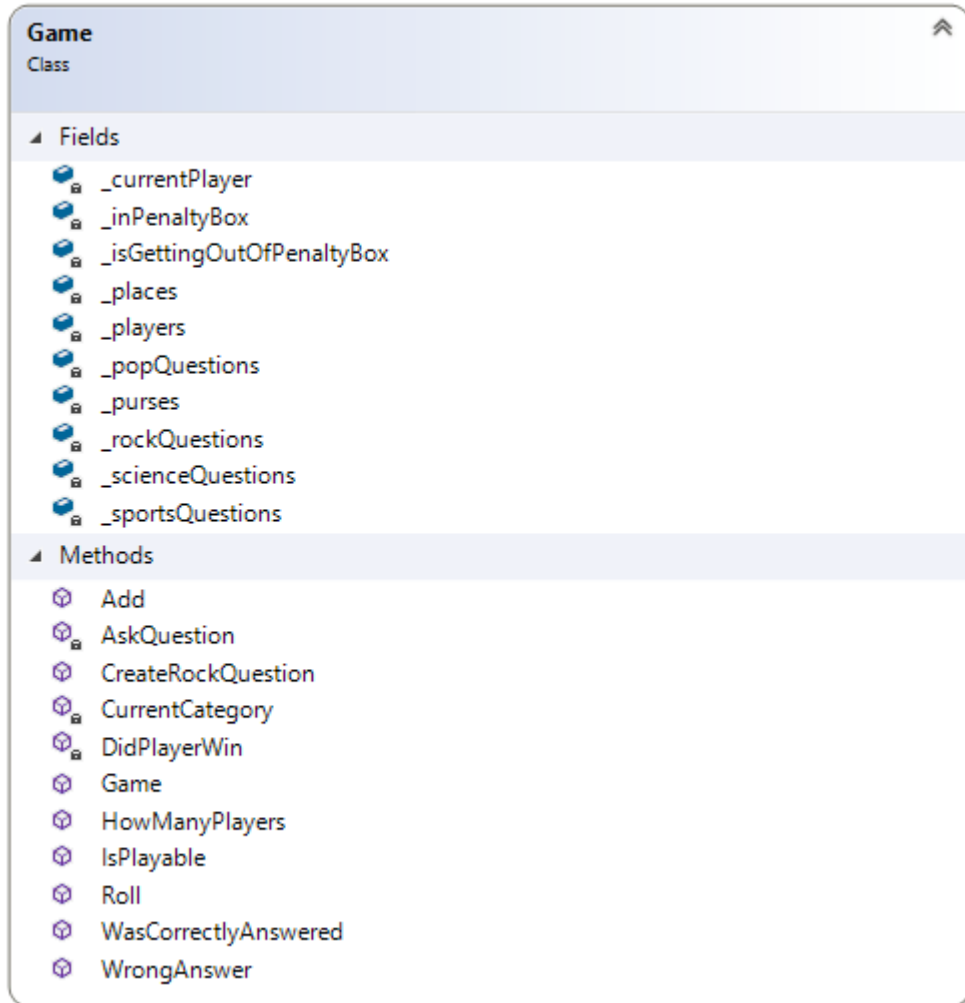
140 public bool WasCorrectlyAnswered()
141 {
142     if (_inPenaltyBox[_currentPlayer])
143     {
144         if (_isGettingOutOfPenaltyBox)
145         {
146             Console.WriteLine("Answer was correct!!!!");
147             _purses[_currentPlayer]++;
148             Console.WriteLine(_players[_currentPlayer]
149                 + " now has "
150                 + _purses[_currentPlayer]
151                 + " Gold Coins.");
152
153             var winner = DidPlayerWin();
154             _currentPlayer++;
155             if (_currentPlayer == _players.Count) _currentPlayer = 0;
156
157             return winner;
158         }
159         else
160         {
161             _currentPlayer++;
162             if (_currentPlayer == _players.Count) _currentPlayer = 0;
163             return true;
164         }
165     }
166     else
167     {
168         Console.WriteLine("Answer was corrent!!!!");
169         _purses[_currentPlayer]++;
170         Console.WriteLine(_players[_currentPlayer]
171             + " now has "
172             + _purses[_currentPlayer]
173             + " Gold Coins.");
174
175         var winner = DidPlayerWin();
176         _currentPlayer++;
177         if (_currentPlayer == _players.Count) _currentPlayer = 0;
178
179         return winner;
180     }

```



```
183     public bool WrongAnswer()
184     {
185         Console.WriteLine("Question was incorrectly answered");
186         Console.WriteLine(_players[_currentPlayer] + " was sent to the penalty box");
187         _inPenaltyBox[_currentPlayer] = true;
188
189         _currentPlayer++;
190         if (_currentPlayer == _players.Count) _currentPlayer = 0;
191         return true;
192     }
193
194
195     private bool DidPlayerWin()
196     {
197         return !(_purses[_currentPlayer] == 6);
198     }
199 }
200
201 }
202
```

Initial Class Diagram of Game Class



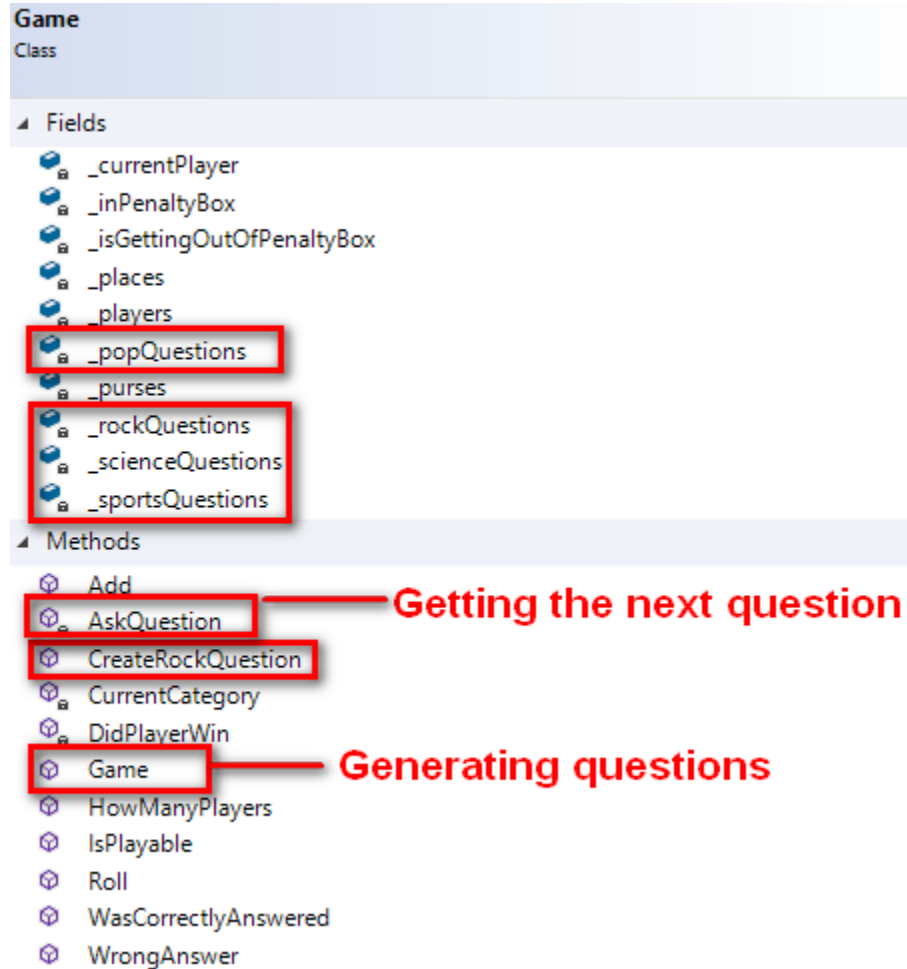
Game class has 3 responsibilities, i.e.:

1. Handling game logic.
2. Generating and managing trivia questions.
3. Managing players.

We need to split the responsibilities by extracting new classes, i.e.: `Players` class and `Questions` class. Thus `Game` class can focus on handling game logic only.

Extract Questions Class:

1. Select Related Methods and Fields



Select any fields and methods which are related to question, i.e.:

- Field:
 - _popQuestions
 - _rockQuestions
 - _scienceQuestions
 - _sportsQuestions
- Method:
 - CreateRockQuestion

Extract Questions Class:

2. Create Questions Class

```
1 namespace Trivia
2 {
3     public class Questions
4     {
5     }
6 }
```

```
1 public class Game
2 {
3     private readonly Questions questions = new Questions();
4     ...
}
```

- Create Questions class to accommodate fields and methods related to generate and manage trivia questions.
- On the other hand, create Questions property in Game class, thus Game class can still access methods and fields of Questions class.

Extract Questions Class:

3. Move Related Fields to Questions Class

```
1 public class Questions
2 {
3     public readonly LinkedList<string> _popQuestions = new LinkedList<string>();
4     public readonly LinkedList<string> _scienceQuestions = new LinkedList<string>();
5     public readonly LinkedList<string> _sportsQuestions = new LinkedList<string>();
6     public readonly LinkedList<string> _rockQuestions = new LinkedList<string>();
7 }
```

- Move `_popQuestions`, `_scienceQuestions`, `_sportsQuestions`, and `_rockQuestions` to Questions class.
- Set their access modifier to public for the time being, thus Game class can still access them. Even though, we have not created any mutator and accessor methods for the aforementioned fields.

Extract Questions Class:

3. Move Related Fields to Questions Class

```
public Game()
{
    for (var i = 0; i < 50; i++)
    {
        _popQuestions.AddLast("Pop Question " + i);
        _scienceQuestions.AddLast(("Science Question " + i));
        _sportsQuestions.AddLast(("Sports Question " + i));
        _rockQuestions.AddLast(CreateRockQuestion(i));
    }
}
```

Update code in Game class thus it can access `_popQuestions`, `_scienceQuestions`, `_sportsQuestions`, and `_rockQuestions` through `Questions` property.



```
public Game()
{
    for (var i = 0; i < 50; i++)
    {
        questions._popQuestions.AddLast("Pop Question " + i);
        questions._scienceQuestions.AddLast(("Science Question " + i));
        questions._sportsQuestions.AddLast(("Sports Question " + i));
        questions._rockQuestions.AddLast(CreateRockQuestion(i));
    }
}
```

Extract Questions Class:

3. Move Related Fields to Questions Class

```
private void AskQuestion()
{
    if (CurrentCategory() == "Pop")
    {
        Console.WriteLine(_popQuestions.First());
        _popQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Science")
    {
        Console.WriteLine(_scienceQuestions.First());
        _scienceQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Sports")
    {
        Console.WriteLine(_sportsQuestions.First());
        _sportsQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Rock")
    {
        Console.WriteLine(_rockQuestions.First());
        _rockQuestions.RemoveFirst();
    }
}
```



```
private void AskQuestion()
{
    if (CurrentCategory() == "Pop")
    {
        Console.WriteLine(questions._popQuestions.First());
        questions._popQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Science")
    {
        Console.WriteLine(questions._scienceQuestions.First());
        questions._scienceQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Sports")
    {
        Console.WriteLine(questions._sportsQuestions.First());
        questions._sportsQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Rock")
    {
        Console.WriteLine(questions._rockQuestions.First());
        questions._rockQuestions.RemoveFirst();
    }
}
```

Extract Questions Class:

4. Move CreateRockQuestion()

```
public Game()
{
    for (var i = 0; i < 50; i++)
    {
        questions._popQuestions.AddLast("Pop Question " + i);
        questions._scienceQuestions.AddLast("Science Question " + i);
        questions._sportsQuestions.AddLast("Sports Question " + i);
        questions._rockQuestions.AddLast(CreateRockQuestion(i));
    }
}

public string CreateRockQuestion(int index)
{
    return "Rock Question " + index;
}
```

- At first, it seems obvious that we have to move CreateRockQuestion() into Questions class. But, after a deeper look, we can see that the method serves no additional purpose other than generate Rock Question.
- On the other hand, generating of Pop Question, Science Question, and Sports Question are handled directly by Game constructor.

Extract Questions Class:

4. Move CreateRockQuestion()

```
public Game()
{
    for (var i = 0; i < 50; i++)
    {
        questions._popQuestions.AddLast("Pop Question " + i);
        questions._scienceQuestions.AddLast("Science Question " + i);
        questions._sportsQuestions.AddLast("Sports Question " + i);
        questions._rockQuestions.AddLast(CreateRockQuestion(i));
    }
}

public string CreateRockQuestion(int index)
{
    return "Rock Question " + index;
}
```



```
public Game()
{
    for (var i = 0; i < 50; i++)
    {
        questions._popQuestions.AddLast("Pop Question " + i);
        questions._scienceQuestions.AddLast("Science Question " + i);
        questions._sportsQuestions.AddLast("Sports Question " + i);
        questions._rockQuestions.AddLast("Rock Question " + i);
    }
}
```

Thus, instead of moving CreateRockQuestion() into Questions class, we can inline the method into Game constructor.

Extract Questions Class:

5. Move Question Generator Logic into Questions Class

```
public void GenerateQuestions()
{
    for (var i = 0; i < 50; i++)
    {
        _popQuestions.AddLast("Pop Question " + i);
        _scienceQuestions.AddLast("Science Question " + i);
        _sportsQuestions.AddLast("Sports Question " + i);
        _rockQuestions.AddLast("Rock Question " + i);
    }
}
```

- Previously, Game constructor is responsible for generating Pop, Science, Sports and Rock Question. But we cannot move constructor into other class. Thus, we move the logic instead.
- In order to accommodate question generator logic, we create a new method in Questions class, which is GenerateQuestions(). And move question generator logic into it.

Extract Questions Class:

5. Move Question Generator Logic into Questions Class

```
public Game()
{
    for (var i = 0; i < 50; i++)
    {
        questions._popQuestions.AddLast("Pop Question " + i);
        questions._scienceQuestions.AddLast("Science Question " + i);
        questions._sportsQuestions.AddLast("Sports Question " + i);
        questions._rockQuestions.AddLast("Rock Question " + i);
    }
}
```



```
public Game()
{
    questions.GenerateQuestions();
}
```

Update code in Game constructor thus it can still generate question through Questions property.

Extract Questions Class:

6. Move Get Next Question Logic into Questions Class

```
private void AskQuestion()
{
    if (CurrentCategory() == "Pop")
    {
        Console.WriteLine(questions._popQuestions.First());
        questions._popQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Science")
    {
        Console.WriteLine(questions._scienceQuestions.First());
        questions._scienceQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Sports")
    {
        Console.WriteLine(questions._sportsQuestions.First());
        questions._sportsQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Rock")
    {
        Console.WriteLine(questions._rockQuestions.First());
        questions._rockQuestions.RemoveFirst();
    }
}
```

- Previously, AskQuestion() in Game class is responsible for:
 1. Getting next question from question list (_popQuestions, _scienceQuestions, _sportsQuestions, and _rockQuestions) according to CurrentCategory,
 2. Writing it on console, and
 3. Removing it from the list.
- We will move get next question logic into Questions class, thus AskQuestion() is only responsible for writing next question to console.

Extract Questions Class:

6. Move Get Next Question Logic into Questions Class

```
public string GetNextQuestion(string Category)
{
    if (Category == "Pop")
    {
        string question = _popQuestions.First();
        _popQuestions.RemoveFirst();
        return question;
    }
    if (Category == "Science")
    {
        string question = _scienceQuestions.First();
        _scienceQuestions.RemoveFirst();
        return question;
    }
    if (Category == "Sports")
    {
        string question = _sportsQuestions.First();
        _sportsQuestions.RemoveFirst();
        return question;
    }
    if (Category == "Rock")
    {
        string question = _rockQuestions.First();
        _rockQuestions.RemoveFirst();
        return question;
    }

    return null;
}
```

- Create a new method in Questions class, which is GetNextQuestion() to accommodate get next question logic from Game class.
- It will return next question back to Game class, since AskQuestion() will write it on console.

Extract Questions Class:

6. Move Get Next Question Logic into Questions Class

```
private void AskQuestion()
{
    if (CurrentCategory() == "Pop")
    {
        Console.WriteLine(questions._popQuestions.First());
        questions._popQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Science")
    {
        Console.WriteLine(questions._scienceQuestions.First());
        questions._scienceQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Sports")
    {
        Console.WriteLine(questions._sportsQuestions.First());
        questions._sportsQuestions.RemoveFirst();
    }
    if (CurrentCategory() == "Rock")
    {
        Console.WriteLine(questions._rockQuestions.First());
        questions._rockQuestions.RemoveFirst();
    }
}
```

Update code in AskQuestion() thus it can still get next question through Questions property.

```
private void AskQuestion()
{
    var question = questions.GetNextQuestion(CurrentCategory());
    Console.WriteLine(question);
}
```

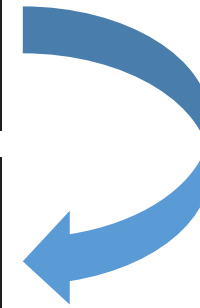


Extract Questions Class:

7. Set access modifier of fields in Questions Class into Private

```
public class Questions
{
    public readonly LinkedList<string> _popQuestions = new LinkedList<string>();
    public readonly LinkedList<string> _scienceQuestions = new LinkedList<string>();
    public readonly LinkedList<string> _sportsQuestions = new LinkedList<string>();
    public readonly LinkedList<string> _rockQuestions = new LinkedList<string>();
}
```

```
public class Questions
{
    private readonly LinkedList<string> _popQuestions = new LinkedList<string>();
    private readonly LinkedList<string> _scienceQuestions = new LinkedList<string>();
    private readonly LinkedList<string> _sportsQuestions = new LinkedList<string>();
    private readonly LinkedList<string> _rockQuestions = new LinkedList<string>();
}
```



In order to keep unwanted access of fields in Questions class, we will set their access modifier into private. Thus, Game class can only access them through method in Questions class, i.e.:

- GenerateQuestions()
- GetNextQuestion()

Extract Players Class:

1. Select Related Methods and Fields

Game
Class

Fields

- _currentPlayer
- _inPenaltyBox**
- isGettingOutOfPenaltyBox
- _places**
- _players**
- _purses**
- questions

Methods

- Add** — **Add Player**
- AskQuestion
- CurrentCategory** — **Get Place**
- DidPlayerWin
- Game
- HowManyPlayers** — **Update Place**
- IsPlayable
- Roll**
- WasCorrectlyAnswered** — **Update Purse**
- WrongAnswer** — **Put in Penalty Box**

Select any fields and methods which are related to player, i.e.:

- Field:
 - _inPenaltyBox
 - _places
 - _players
 - _purses
- Method:
 - Add()
 - HowManyPlayers()

Extract Players Class:

2. Create Players Class

```
namespace Trivia
{
    public class Players
    {
    }
}
```

```
public Game()
{
    questions.GenerateQuestions();
}
```



```
private readonly Players players;
public Game(Players players)
{
    this.players = players;
    questions.GenerateQuestions();
}
```

- Create Players class to accommodate fields and methods related to manage players.
- On the other hand, create Players property in Game class, thus Game class can still access methods and fields of Players class.
- Game constructor will inject Players object into Players property. Thus we do not need to create new Players each time we start a game.

Extract Players Class:


3. Move Related Fields into Players Class

```
public class Game
{
    private readonly List<string> _players = new List<string>();

    private readonly int[] _places = new int[6];
    private readonly int[] _purses = new int[6];

    private readonly bool[] _inPenaltyBox = new bool[6];
}
```

```
public class Players
{
    public readonly List<string> _playerNames = new List<string>();
    public readonly int[] _places = new int[6];
    public readonly int[] _purses = new int[6];
    public readonly bool[] _inPenaltyBox = new bool[6];
}
```



Move Fields into
Players class

Previously Game class has `_players` field to store list of players' names. Since we have Players class, we can move it to Players class instead. To make it more meaningful, we will rename it into `_playerNames`.

Extract Players Class:

3. Move Related Fields into Players Class

```
public bool Add(string playerName)
{
    _players.Add(playerName);
    _places[HowManyPlayers()] = 0;
    _purses[HowManyPlayers()] = 0;
    _inPenaltyBox[HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + _players.Count);
    return true;
}
```

```
public bool Add(string playerName)
{
    players._playerNames.Add(playerName);
    players._places[HowManyPlayers()] = 0;
    players._purses[HowManyPlayers()] = 0;
    players._inPenaltyBox[HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + HowManyPlayers());
    return true;
}
```

Update code in Game class
thus it can access
_playerNames, _places,
_purses, _inPenaltyBox
through Players property.

```
public int HowManyPlayers()
{
    return _players.Count;
}
```

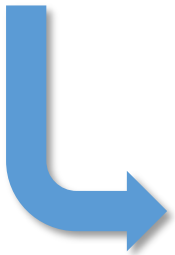
```
public int HowManyPlayers()
{
    return players._playerNames.Count;
}
```



Extract Players Class:

3. Move Related Fields into Players Class

```
private string CurrentCategory()  
{  
    if (_places[_currentPlayer] == 0) return "Pop";  
    if (_places[_currentPlayer] == 4) return "Pop";  
    if (_places[_currentPlayer] == 8) return "Pop";  
    if (_places[_currentPlayer] == 1) return "Science";  
    if (_places[_currentPlayer] == 5) return "Science";  
    if (_places[_currentPlayer] == 9) return "Science";  
    if (_places[_currentPlayer] == 2) return "Sports";  
    if (_places[_currentPlayer] == 6) return "Sports";  
    if (_places[_currentPlayer] == 10) return "Sports";  
    return "Rock";  
}
```



```
private string CurrentCategory()  
{  
    if (players._places[_currentPlayer] == 0) return "Pop";  
    if (players._places[_currentPlayer] == 4) return "Pop";  
    if (players._places[_currentPlayer] == 8) return "Pop";  
    if (players._places[_currentPlayer] == 1) return "Science";  
    if (players._places[_currentPlayer] == 5) return "Science";  
    if (players._places[_currentPlayer] == 9) return "Science";  
    if (players._places[_currentPlayer] == 2) return "Sports";  
    if (players._places[_currentPlayer] == 6) return "Sports";  
    if (players._places[_currentPlayer] == 10) return "Sports";  
    return "Rock";  
}
```

Update code in Game class thus it can access `_playerNames`, `_places`, `_purses`, `_inPenaltyBox` through `Players` property.

Extract Players Class:

4. Move HowManyPlayers() into Players Class

```
public bool Add(string playerName)
{
    players._playerNames.Add(playerName);
    players._places[HowManyPlayers()] = 0;
    players._purses[HowManyPlayers()] = 0;
    players._inPenaltyBox[HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + HowManyPlayers());
    return true;
}
```

```
public bool Add(string playerName)
{
    players._playerNames.Add(playerName);
    players._places[players.HowManyPlayers()] = 0;
    players._purses[players.HowManyPlayers()] = 0;
    players._inPenaltyBox[players.HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + players.HowManyPlayers());
    return true;
}
```

Update code in Game class
thus it can access
HowManyPlayers() through
Players property.

```
public bool IsPlayable()
{
    return (HowManyPlayers() >= 2);
}
```

```
public bool IsPlayable()
{
    return (players.HowManyPlayers() >= 2);
}
```

Extract Players Class:

4. Move HowManyPlayers() into Players Class

```
public int HowManyPlayers()  
{  
    return players._playerNames.Count;  
}
```



```
public int HowManyPlayers()  
{  
    return _playerNames.Count;  
}
```

Remove Players property from HowManyPlayers(), because it is moved into Players class. Thus HowManyPlayers() can access _playerNames directly.

Extract Players Class:

4. Move HowManyPlayers() into Players Class

```
public bool Add(string playerName)
{
    players._playerNames.Add(playerName);
    players._places[HowManyPlayers()] = 0;
    players._purses[HowManyPlayers()] = 0;
    players._inPenaltyBox[HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + HowManyPlayers());
    return true;
}
```

```
public bool Add(string playerName)
{
    players._playerNames.Add(playerName);
    players._places[players.HowManyPlayers()] = 0;
    players._purses[players.HowManyPlayers()] = 0;
    players._inPenaltyBox[players.HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + players.HowManyPlayers());
    return true;
}
```

Update code in Game class thus it can access HowManyPlayers() through Players property.

```
public bool IsPlayable()
{
    return (HowManyPlayers() >= 2);
}
```

```
public bool IsPlayable()
{
    return (players.HowManyPlayers() >= 2);
}
```

Extract Players Class:

4. Move Add() into Players Class

```
public bool Add(string playerName)
{
    players._playerNames.Add(playerName);
    players._places[players.HowManyPlayers()] = 0;
    players._purses[players.HowManyPlayers()] = 0;
    players._inPenaltyBox[players.HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + players.HowManyPlayers());
    return true;
}
```

```
public bool Add(string playerName)
{
    _playerNames.Add(playerName);
    _places[HowManyPlayers()] = 0;
    _purses[HowManyPlayers()] = 0;
    _inPenaltyBox[HowManyPlayers()] = false;

    Console.WriteLine(playerName + " was added");
    Console.WriteLine("They are player number " + HowManyPlayers());
    return true;
}
```

Remove Players property from Add(), because it is moved into Players class. Thus Add() can access _playerNames, _places, _purses, _inPenaltyBox, and HowManyPlayers() directly.




Extract Players Class:

5. Encapsulate Fields in Players Class

```
public readonly List<string> _playerNames = new List<string>();  
public readonly int[] _places = new int[6];  
public readonly int[] _purses = new int[6];  
public readonly bool[] _inPenaltyBox = new bool[6];
```

```
private readonly List<string> _playernames = new List<string>();  
private readonly int[] _places = new int[6];  
private readonly int[] _purses = new int[6];  
private readonly bool[] _inPenaltyBox = new bool[6];
```



In order to keep unwanted access of fields in Players class, we will set their access modifier into private. Thus, we need to create mutator and/or accessor methods for each field.

Extract Players Class:

5. Encapsulate Fields in Players Class

```
public int GetPlace(int playerNumber)
{
    return _places[playerNumber];
}
public void AddToPlace(int playerNumber, int addAmount)
{
    _places[playerNumber] += addAmount;
}
```

GetPlace() is accessor method for _places. Whereas AddToPlace() is mutator method for _places.

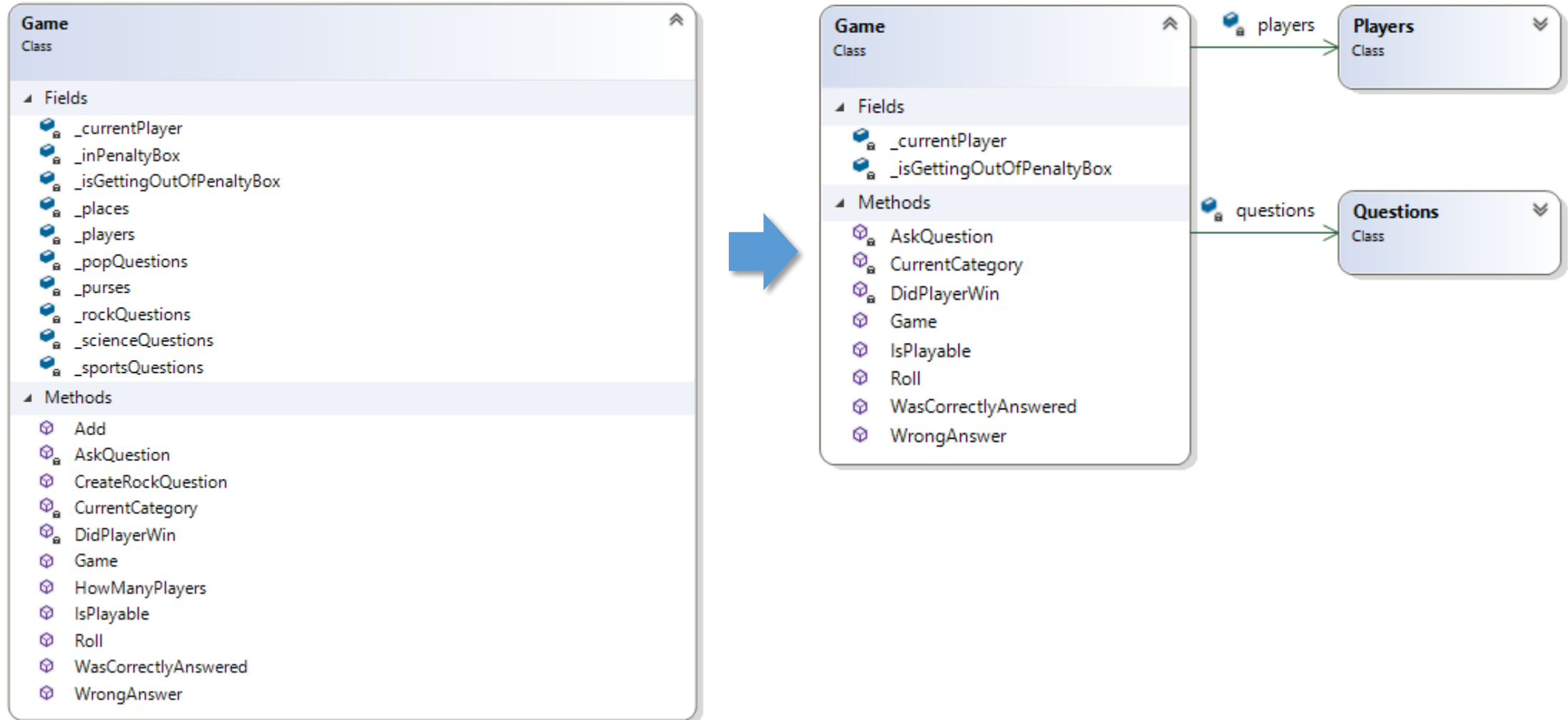
Update code in Game Class accordingly, thus it can still access fields in Player Class (_playerNames, _places, _purses, _inPenaltyBox) indirectly through their accessor and mutator methods.

```
_places[_currentPlayer] = _places[_currentPlayer] + roll;
if (_places[_currentPlayer] > 11) _places[_currentPlayer] = _places[_currentPlayer] - 12;
```

```
players.AddToPlace(_currentPlayer, roll);
if (players.GetPlace(_currentPlayer) > 11) players.AddToPlace(_currentPlayer, -12);
```



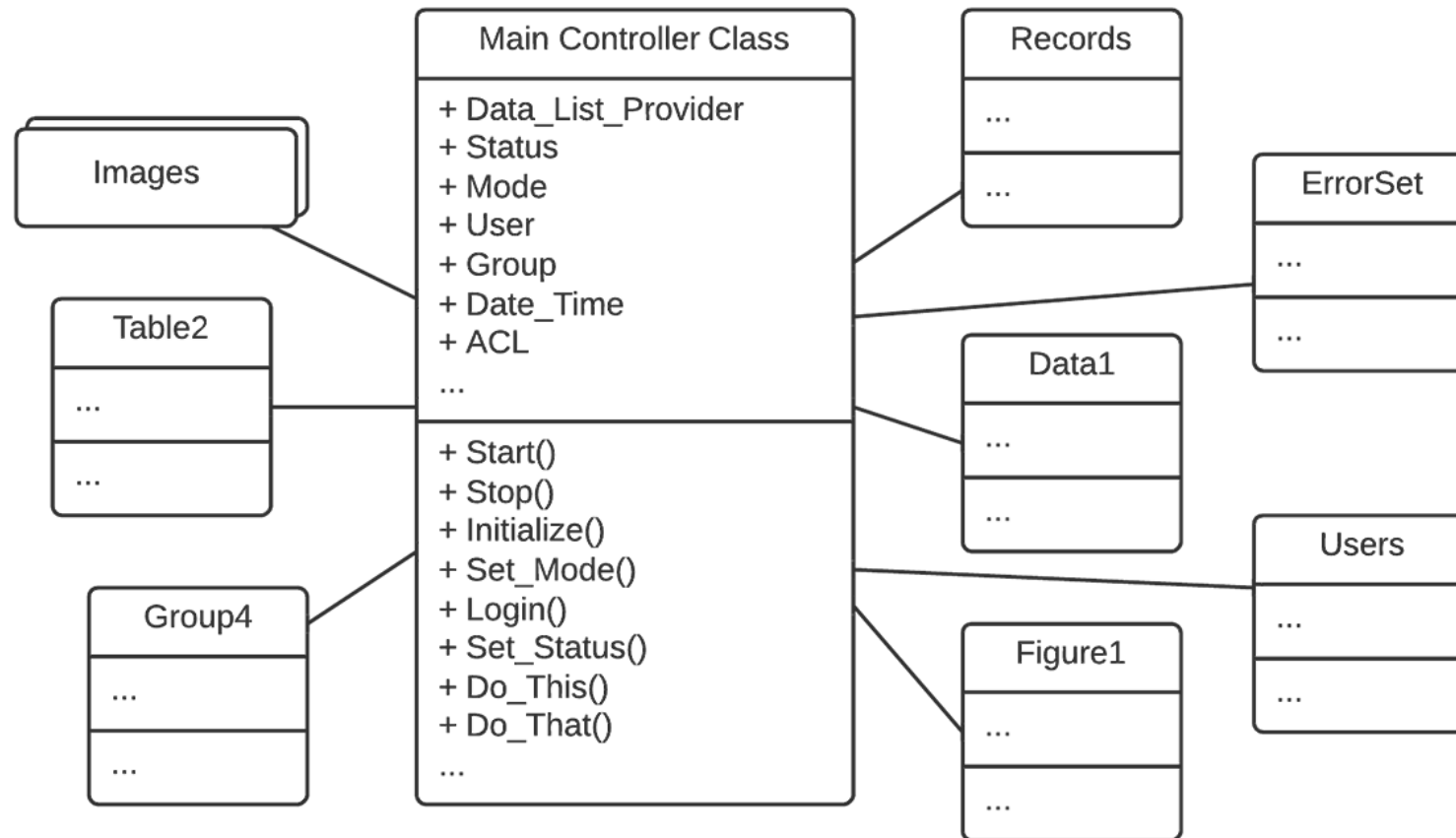
Revised Class Diagram of Game Class



God Class

- God class belongs to Software Development Anti-pattern and is known as the blob or Winnebago.
- It is characterized by a class diagram composed of a single complex controller class surrounded by simple data classes.
- The key problem here is that the majority of the responsibilities are allocated to a single class.





The Blob contains the majority of the process, and the other objects contain the data. Architectures with the Blob have separated process from data;

God Class: Symptoms

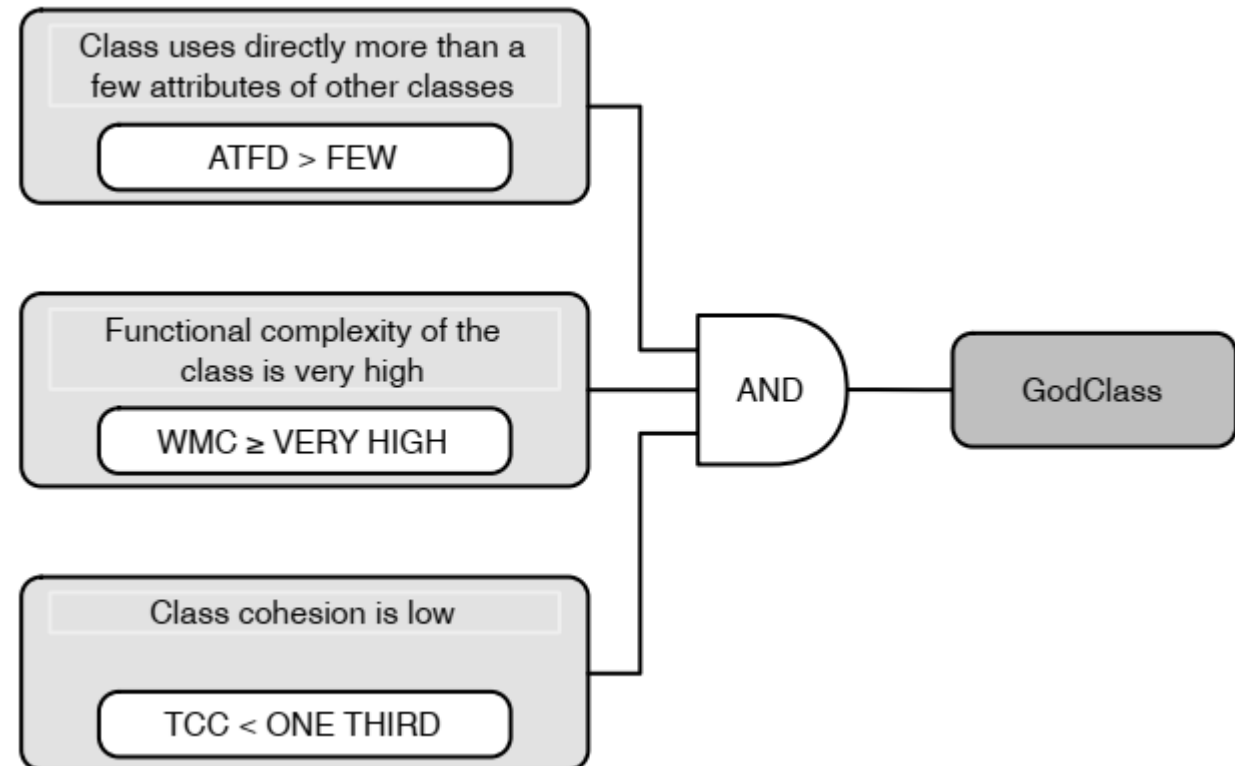
- A class with 60 or more attributes and operations usually indicates the presence of the Blob.
- A disparate collection of unrelated attributes and operations encapsulated in a single class (Lack of cohesiveness).
- A single controller class with associated simple, data-object classes.
- The single controller class often nearly encapsulates the applications entire functionality (Absence of object-oriented design).

God Class: Exceptions

- The Blob is acceptable when wrapping legacy systems.
- Legacy system is an old method, technology, computer system, or application program, "of, relating to, or being a previous or outdated computer system," yet still in use.

God Class

- A God Class features a high complexity, low inner-class cohesion, and heavy access to data of foreign classes.
- Thus, Weighted Method Count (WMC), Tight Class Cohesion (TCC), and Access To Foreign Data (ATFD) can be used to detect God Class

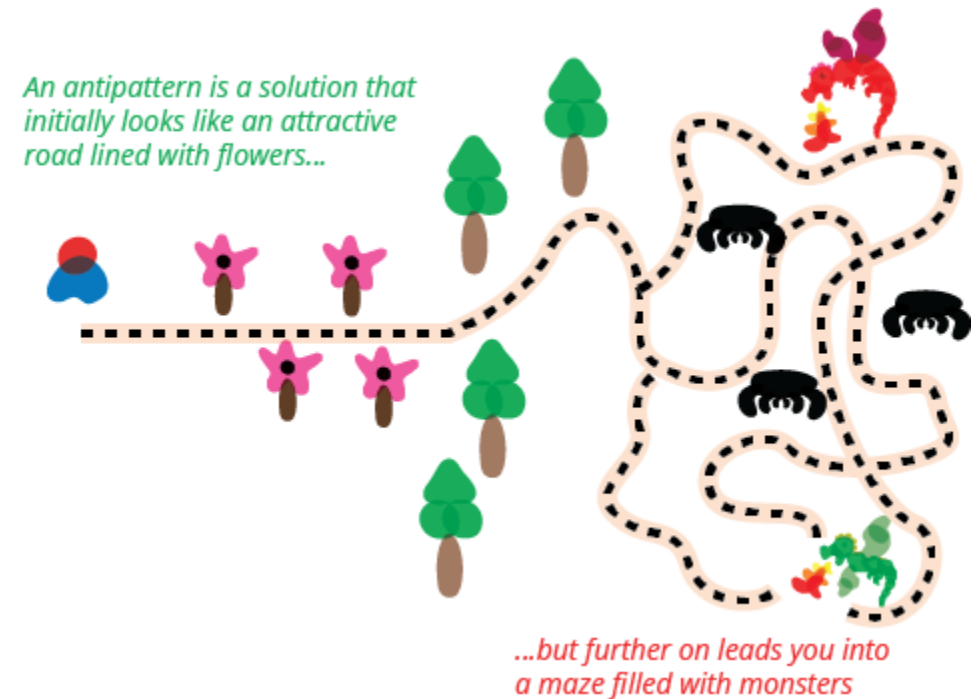


God Class

- Weighted Method Count ($WMC(C)$) is the sum of the cyclomatic complexity of all methods in C .
- Tight Class Cohesion ($TCC(C)$) is the relative number of directly connected methods in C . Two methods are directly connected if they access the same instance variables of C .
- Access To Foreign Data ($ATFD(C)$) is the number of attributes of foreign classes accessed directly by class C or via accessor methods

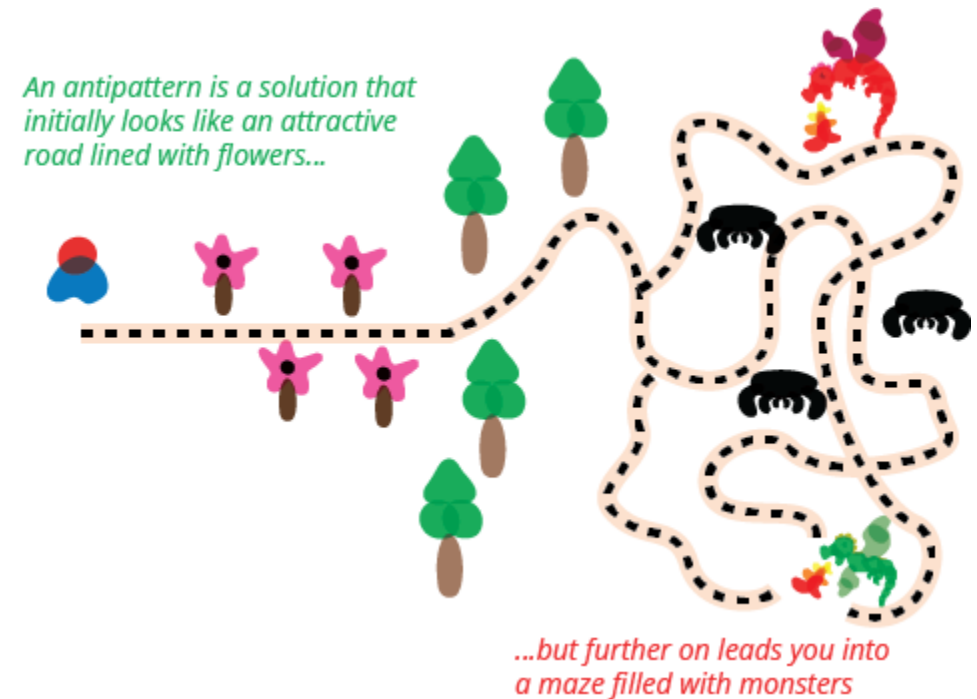
Code Smell vs Anti-pattern

- *An anti-pattern is just like a pattern, except that instead of a solution it gives something that looks superficially like a solution but isn't one - Andrew Koenig*
- The same solution can be a good pattern in some contexts and an antipattern in others. The value of a solution depends on the context that you use it.



Code Smell vs Anti-pattern

- Software anti-pattern can be categorized into 3 aspects, which are:
 1. Software Development AntiPatterns
 2. Software Architecture AntiPatterns
 3. Software Project Management AntiPatterns



Code Smell vs Anti-pattern

- A smell is by definition something that's quick to spot.
- Smells don't always indicate a problem. You have to look deeper to see if there is an underlying problem there.
- Smells aren't inherently bad on their own - they are often an indicator of a problem rather than the problem themselves.

Automated Code Smell Detection Tools

- Checkstyle
- Jdeodorant
- PMD
- InFusion
- iPlasma
- StenchBlossom
- JSpIRIT



References

- Martin, Robert C. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson. 2008.
- Fowler, Martin. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
- Lanza, M., & Marinescu, R. Object-Oriented Metrics in Practice. Springer. 2006.
- <https://refactoring.guru/>
- <https://sourcemaking.com/>
- <https://martinfowler.com/bliki/AntiPattern.html>
- <https://martinfowler.com/bliki/CodeSmell.html>
- <https://makolyte.com/refactoring-the-large-class-code-smell/>



bridge to the future

<http://www.eepis-its.edu>