Refactoring

Object-oriented software design

Dr. Balázs Simon

BME, IIT

True or false

True or false

Is the following statement true or false?

The goal of refactoring is to fix bugs.

True or false

Is the following statement true or false?

Solution: False

The goal of refactoring is to fix bugs.

True or false

Is the following statement true or false?

We need tests with good code coverage to be able to refactor.

True or false

Is the following statement true or false?

Solution: True

We need tests with good code coverage to be able to refactor.

True or false

Is the following statement true or false?

It is recommended to do multiple refactoring transformations at once in order to spare time.

True or false

Is the following statement true or false?

Solution: False

It is recommended to do multiple refactoring transformations at once in order to spare time.

True or false

Is the following statement true or false?

A code smell indicates a bug in the software.

True or false

Is the following statement true or false?

Solution: False

A code smell indicates a bug in the software.

True or false

Is the following statement true or false?

Refactoring improves the structure of the code.

True or false

Is the following statement true or false?

Solution: True

Refactoring improves the structure of the code.

True or false

Is the following statement true or false?

There exist refactoring operations which are inverses of each other.

True or false

Is the following statement true or false?

Solution: True

There exist refactoring operations which are inverses of each other.

Matching

Match the following refactoring patterns with the most appropriate code smell!

1. Replace magic numbers with symbolic constant
2. Extract method
3. Preserve whole object
4. Decompose conditional
5. Replace array with object
6. Hide delegate
7. Inline class

A. Temporary field
B. Speculative generality
C. Shotgun surgery
D. Message chains
E. Lazy class
F. Duplicated code
G. Primitive obsession
H. Switch statement
I. Long parameter lists
J. Refused bequest
K. Divergent change

Match the following refactoring patterns with the most appropriate code smell!

1. Replace magic numbers with symbolic constant
2. Extract method
3. Preserve whole object
4. Decompose conditional
5. Replace array with object
6. Hide delegate
7. Inline class

A. Temporary field
B. Speculative generality
C. Shotgun surgery
D. Message chains
E. Lazy class
F. Duplicated code
G. Primitive obsession
H. Switch statement
I. Long parameter lists
J. Refused bequest
K. Divergent change

Solution:

1	2	3	4	5	6	7
C	F	I	H	G	D	E

Match the following refactoring patterns and code smells with the most appropriate OO design principle!

1. Smell: Duplicated code
2. Smell: Large class
3. Smell: Message chains
4. Refactor: Replace type code with class
5. Refactor: Separate domain from presentation
6. Refactor: Extract interface
7. Smell: Speculative generality

A. SDP
B. OCP
C. SRP
D. ADP
E. DRY
F. YAGNI
G. ISP
H. LoD

Match the following refactoring patterns and code smells with the most appropriate OO design principle!

1. Smell: Duplicated code
2. Smell: Large class
3. Smell: Message chains
4. Refactor: Replace type code with class
5. Refactor: Separate domain from presentation
6. Refactor: Extract interface
7. Smell: Speculative generality

A. SDP
B. OCP
C. SRP
D. ADP
E. DRY
F. YAGNI
G. ISP
H. LoD

Solution:

1	2	3	4	5	6	7
E	C	H	B	A	G	F

Match the following OO design heuristics with the most appropriate refactoring pattern or code smell!

1. If you need to override a method with an empty implementation the inheritance hierarchy is wrong
2. Never encode behavior with enum or int values, use polymorphism instead
3. Keep the number of public methods in a class minimal
4. Do not use non-public members of another class
5. Minimize the number of different method calls between collaborating classes

A. Smell: Comments
B. Refactor: Replace type code with class
C. Smell: Inappropriate intimacy
D. Refactor: Remove control flag
E. Refactor: Hide method
F. Smell: Temporary field
G. Smell: Refused bequest
H. Refactor: Introduce explaining variable
I. Refactor: Substitute algorithm
J. Refactor: Introduce assertion
K. Smell: Feature envy
L. Refactor: Replace constructor with factory method
M. Smell: Long parameter lists

Match the following OO design heuristics with the most appropriate refactoring pattern or code smell!

1. If you need to override a method with an empty implementation the inheritance hierarchy is wrong
2. Never encode behavior with enum or int values, use polymorphism instead
3. Keep the number of public methods in a class minimal
4. Do not use non-public members of another class
5. Minimize the number of different method calls between collaborating classes

A. Smell: Comments
B. Refactor: Replace type code with class
C. Smell: Inappropriate intimacy
D. Refactor: Remove control flag
E. Refactor: Hide method
F. Smell: Temporary field
G. Smell: Refused bequest
H. Refactor: Introduce explaining variable
I. Refactor: Substitute algorithm
J. Refactor: Introduce assertion
K. Smell: Feature envy
L. Refactor: Replace constructor with factory method
M. Smell: Long parameter lists

Solution:

1	2	3	4	5
G	B	E	C	K

Match the following OO design heuristics with the most appropriate refactoring pattern or code smell!

1. Move common data and behavior as high as possible in the inheritance hierarchy
2. Avoid classes with only accessor methods
3. Attributes should be private
4. The view should be dependent on the model and not the other way around
5. Model optional elements as containment and never as inheritance

A. Smell: Temporary field
B. Refactor: Introduce explaining variable
C. Refactor: Replace constructor with factory method
D. Refactor: Separate domain from presentation
E. Refactor: Introduce null object
F. Refactor: Pull up method
G. Refactor: Introduce assertion
H. Refactor: Encapsulate field
I. Refactor: Substitute algorithm
J. Smell: Long parameter lists
K. Smell: Data class
L. Refactor: Remove control flag
M. Smell: Comments

Match the following OO design heuristics with the most appropriate refactoring pattern or code smell!

1. Move common data and behavior as high as possible in the inheritance hierarchy
2. Avoid classes with only accessor methods
3. Attributes should be private
4. The view should be dependent on the model and not the other way around
5. Model optional elements as containment and never as inheritance

A. Smell: Temporary field
B. Refactor: Introduce explaining variable
C. Refactor: Replace constructor with factory method
D. Refactor: Separate domain from presentation
E. Refactor: Introduce null object
F. Refactor: Pull up method
G. Refactor: Introduce assertion
H. Refactor: Encapsulate field
I. Refactor: Substitute algorithm
J. Smell: Long parameter lists
K. Smell: Data class
L. Refactor: Remove control flag
M. Smell: Comments

Solution:

1	2	3	4	5
F	K	H	D	E

Code refactoring

Task

- An example with code smells
- Find the code smells
- Find the corresponding refactoring patterns
- Make the example correct using the refactoring patterns

Example with code smells

```
public string[] GetPerson()
{
    string[] person = new string[4];
    person[0] = "John";
    person[1] = "Smith";
    person[2] = "07/03/1982";
    person[3] = "London";
    return person;
}
```

Code smell: Primitive obsession

Refactoring: Replace array with object

Solution

```
public Person GetPerson()
{
    return new Person("John", "Smith", new DateTime(1982,3,7), "London");
}

public class Person
{
    public Person(string firstName, string lastName,
                  DateTime dateOfBirth, string placeOfBirth)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
        this.DateOfBirth = dateOfBirth;
        this.PlaceOfBirth = placeOfBirth;
    }

    public string FirstName { get; }
    public string LastName { get; }
    public DateTime DateOfBirth { get; }
    public string PlaceOfBirth { get; }
}
```

Example with code smells

```
public double PotentialEnergy(double mass, double height)
{
    return mass * 9.81 * height;
}
```

Code smell: Shotgun surgery (excessive use of literals)

Refactoring: Replace magic numbers with symbolic constant

Solution

```
public const double GravitationalConstant = 9.81;

public double PotentialEnergy(double mass, double height)
{
    return mass * GravitationalConstant * height;
}
```

Example with code smells

```
public void Turn()
{
    for (int i = 0; i < MaxPlayers; i++)
    {
        if (players[i] != null)
        {
            players[i].Play();
        }
    }
}
```

Code smell: Switch statements (Null checking)

Refactoring: Introduce Null Object

Solution

```
public void Turn()
{
    for (int i = 0; i < MaxPlayers; i++)
    {
        players[i].Play();
    }
}


public class NullPlayer : Player
{
    public override void Play()
    {
        // nop
    }
}
```

Example with code smells

```
public void HandlePayment(double price)
{
    if (IsSpecialDeal())
    {
        double total = price * 0.95;
        Send(total);
    }
    else
    {
        double total = price * 0.98;
        Send(total);
    }
}
```

Code smells:
Duplicated code
Shotgun surgery (excessive use of literals)

Refactorings:
Consolidate duplicate conditional fragments
Replace magic numbers with symbolic constant

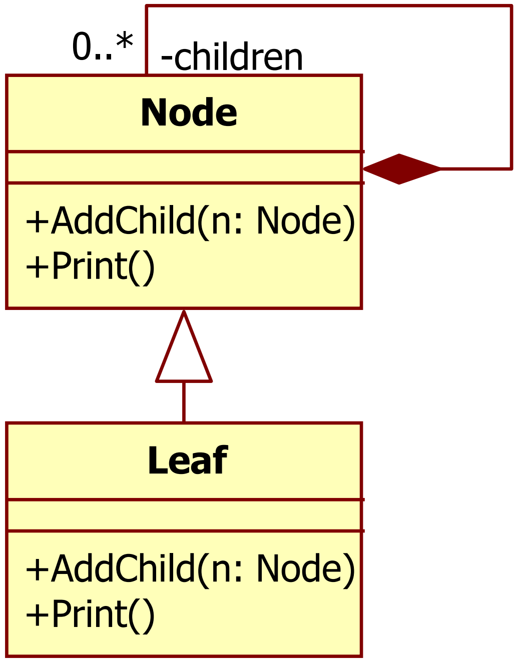
Solution

```
public const SpecialDealMultiplier = 0.95;
public const NormalDealMultiplier = 0.98;

public void HandlePayment(double price)
{
    double multiplier;
    if (IsSpecialDeal()) multiplier = SpecialDealMultiplier;
    else multiplier = NormalDealMultiplier;

    double total = price * multiplier;
    Send(total);
}
```

Example with code smells



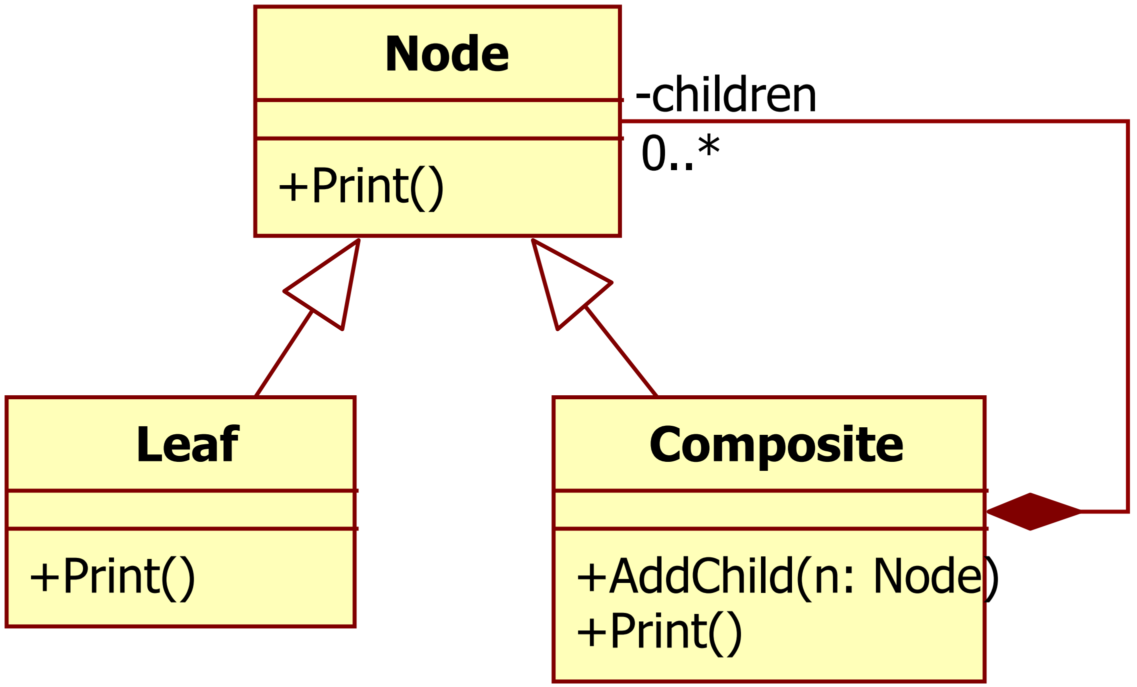
```
 public class Node {
     public virtual void AddChild(Node n) {
         this.children.Add(n);
     }
 }

public class Leaf : Node {
    public override void AddChild(Node n) {
        // nop
    }
}
```

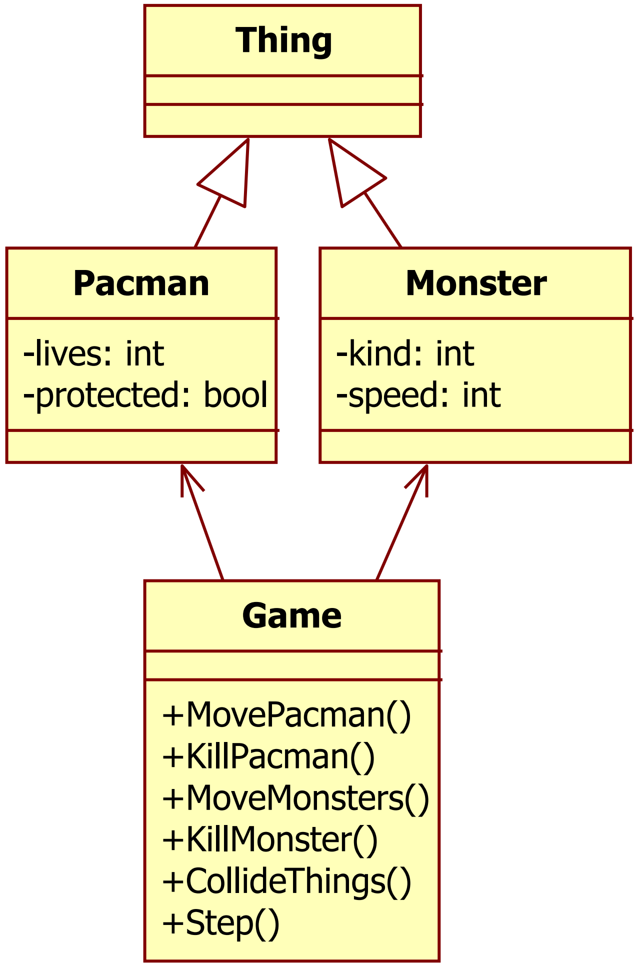
Code smell: Refused bequest

Refactorings:
Reorder the inheritance hierarchy
Push down method

Solution


Node
+Print()
-children
0..*
Leaf
+Print()
Composite
+AddChild(n: Node)
+Print()

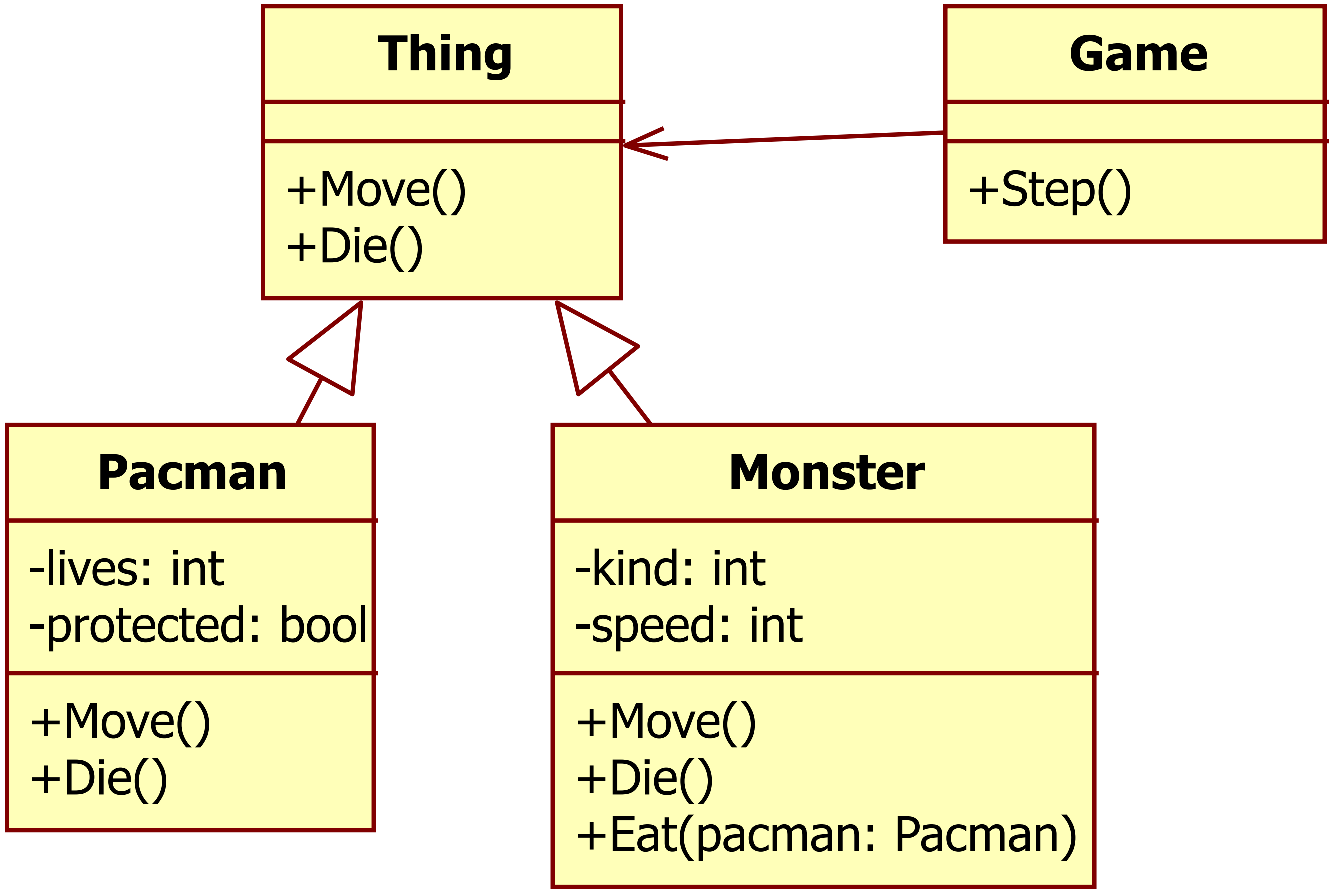
Example with code smells



Code smells:
Data class
Large class

Refactorings:
Move method

Solution


Thing
+Move()
+Die()
Game
+Step()
Pacman
-lives: int
-protected: bool
+Move()
+Die()
Monster
-kind: int
-speed: int
+Move()
+Die()
+Eat(pacman: Pacman)

Example with code smells

```
public void HandleOrder(int orderId, double total, Item[] items, string address)
{
    this.CheckOrder(orderId, total, items);
    this.SendOrder(orderId, total, items, address);
}

private void CheckOrder(int orderId, double total, Item[] items)
{
    // ...
}

private void SendOrder(int orderId, double total, Item[] items, string address)
{
    // ...
}
```

Code smells:
Data clumps
Long parameter lists

Refactoring:
Introduce parameter object

Solution

```
public void HandleOrder(Order order)
{
    this.CheckOrder(order);
    this.SendOrder(order);
}

private void CheckOrder(Order order)
{
    // ...
}

private void SendOrder(Order order)
{
    // ...
}
```

```
public class Order
{
    public int Id { get; }
    public double Total { get; }
    public Item[] Items { get; }
    public string Address { get; }
}
```

Refactoring:
Move method

Solution

```
public void HandleOrder(Order order)
{
    order.Check();
    order.Send();
}

public class Order
{
    public int Id { get; }
    public double Total { get; }
    public Item[] Items { get; }
    public string Address { get; }

    public void Check()
    {
        // ...
    }
    public void Send()
    {
        // ...
    }
}
```

Example with code smells

```
class Pacman {
  void Step() {
    Field next = field.GetNext();
    if (next.IsFree) {
      next.Accept(this);
    } else {
      Thing other = next.GetThing();
      other.Collide(this);
    }
  }
}
```

Code smell:
Feature envy

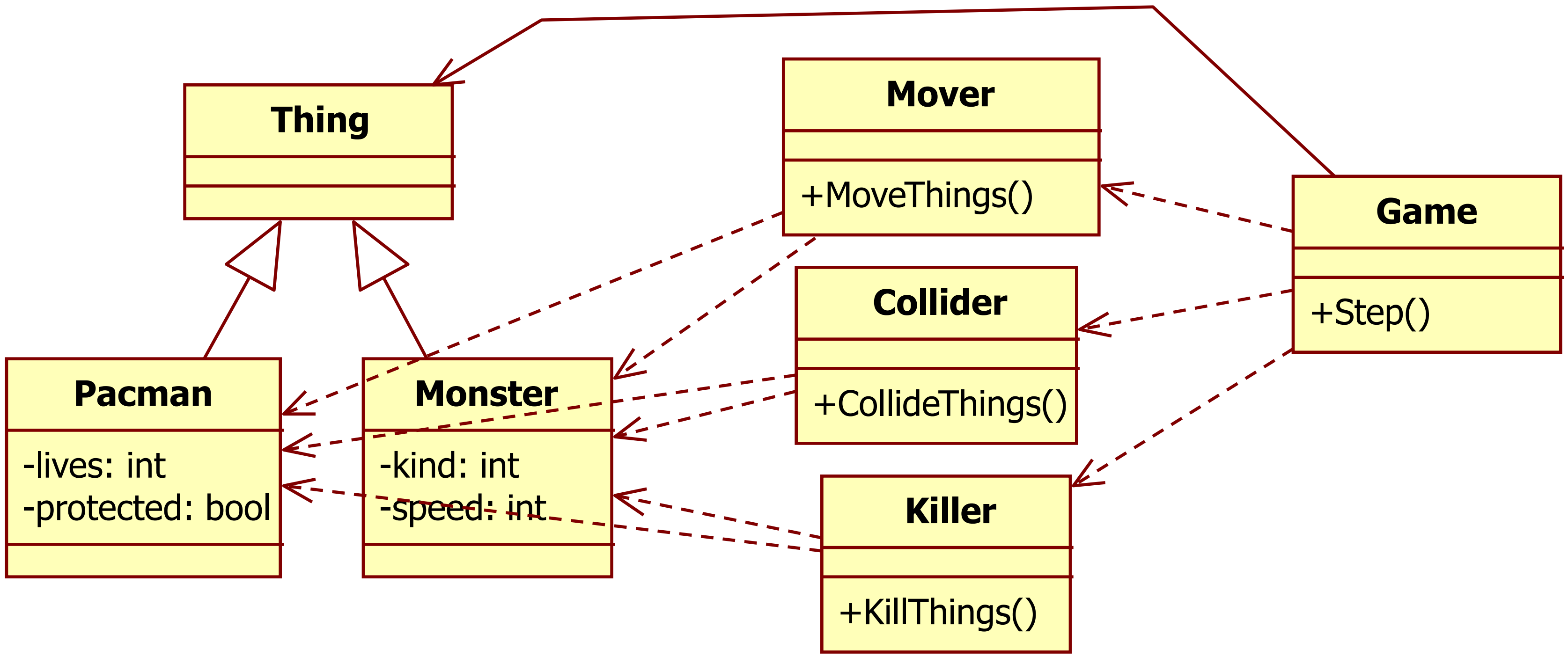
Refactoring:
Extract the relevant part of the method and put into the other class

Solution

```
class Pacman {
  void Step() {
    Field next = field.GetNext();
    field.Remove(this);
    next.Accept(this);
  }
}

class Field {
  void Accept(Thing t) {
    if (this.thing != null) {
      this.thing.Collide(t);
    } else {
      this.thing = t;
    }
  }
}
```

Example with code smells



Code smells:

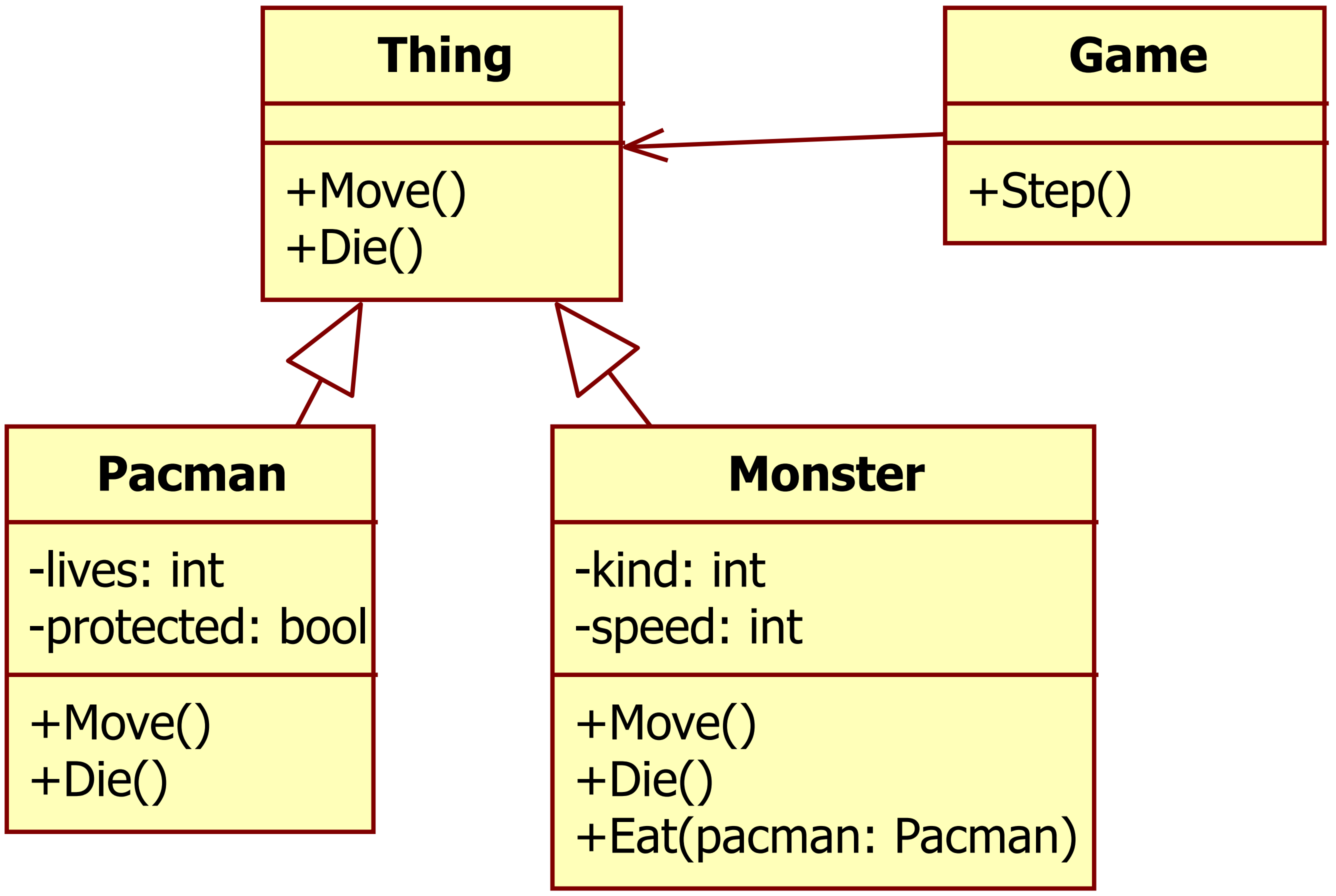
Data class

Lazy class

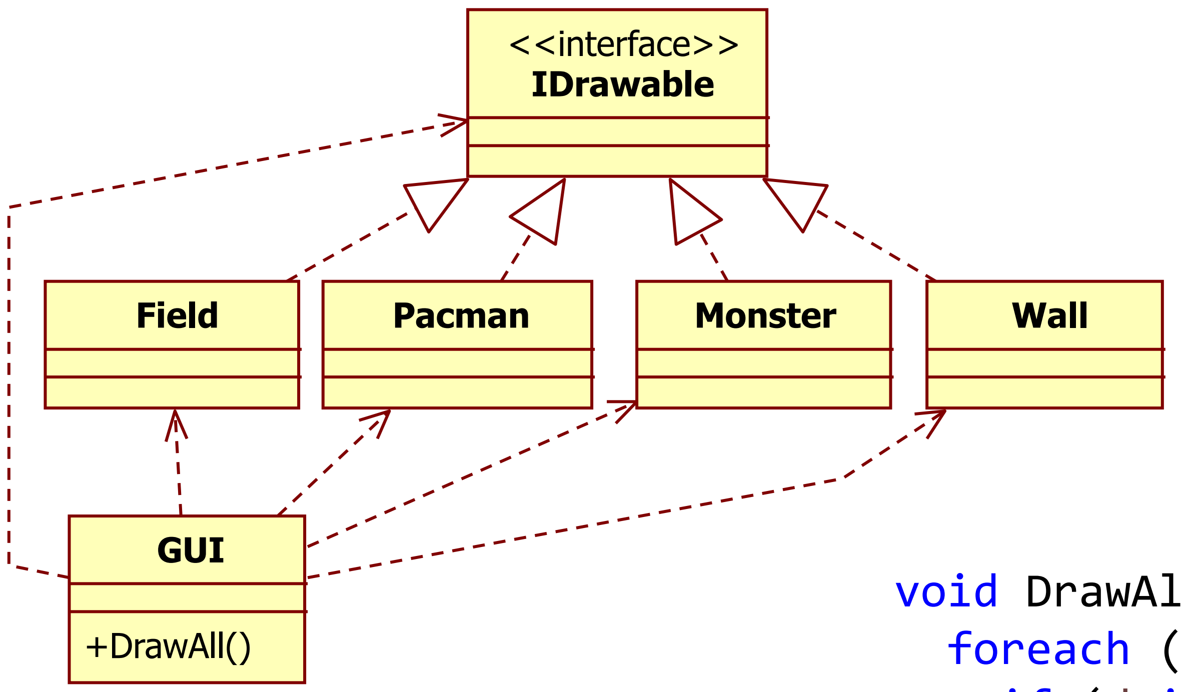
Refactoring:

Move method

Solution


Thing
+Move()
+Die()
Game
+Step()
Pacman
-lives: int
-protected: bool
+Move()
+Die()
Monster
-kind: int
-speed: int
+Move()
+Die()
+Eat(pacman: Pacman)

Example with code smells

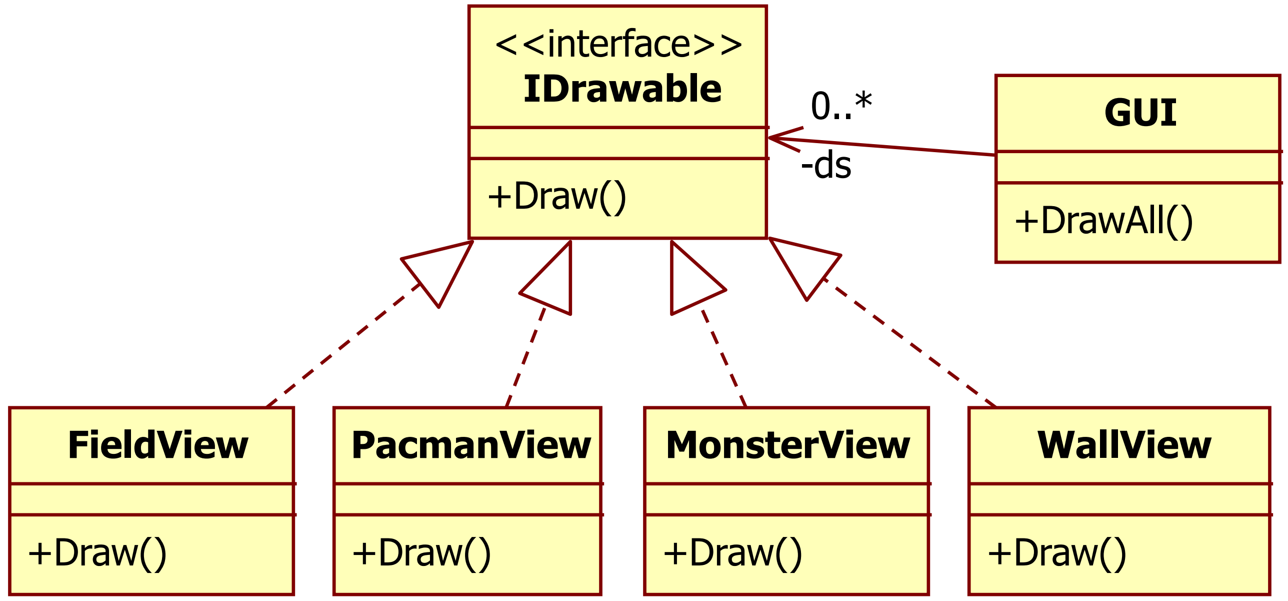


```
void DrawAll(List<IDrawable> ds) {
  foreach (var d in ds) {
    if (d is Field) {
      // ...draw field...
    } else if (d is Pacman) {
      // ...draw pacman...
    } else if ...
  }
}
```

Code smells:
Downcasting
Switch statement

Refactoring:
Replace conditional with polymorphism

Solution



```
void DrawAll(List<IDrawable> ds) {
  foreach (var d in ds) {
    d.Draw();
  }
}
```

Example with code smells

```
public class Graph
{
    private List<Node> nodes;
    private List<Edge> edges;

    private HashSet<Node> visited;

    public Node[] BFS(Node start)
    {
        this.visited = new HashSet<Node>();
        this.VisitNode(start);
        return this.visited.ToArray();
    }

    private void VisitNode(Node current)
    {
        visited.Add(current);
        // ...
    }
}
```

Code smell:
Temporary field

Refactoring:
Add parameter
OR:
Extract class

Solution with “Add parameter”

```
public class Graph
{
    private List<Node> nodes;
    private List<Edge> edges;

    public Node[] BFS(Node start)
    {
        HashSet<Node> visited = new HashSet<Node>();
        this.VisitNode(start, visited);
        return visited.ToArray();
    }

    private void VisitNode(Node current, HashSet<Node> visited)
    {
        visited.Add(current);
        // ...
    }
}
```

Solution with “Extract class”

```
public class Graph {
    private List<Node> nodes;
    private List<Edge> edges;
    public Node[] BFS(Node start) {
        return new BfsAlgorithm(this).Execute(start);
    }
}
public class BfsAlgorithm {
    private Graph graph;
    private HashSet<Node> visited = new HashSet<Node>();

    public BfsAlgorithm(Graph graph) { this.graph = graph; }

    public Node[] Execute(Node start) {
        this.VisitNode(start);
        return visited.ToArray();
    }
    private void VisitNode(Node current) {
        visited.Add(current);
        // ...
    }
}
```