b
UNIVERSITÄT
BERN

Software Design Patterns

Pattern types

Creational Patterns

Behavioural Patterns

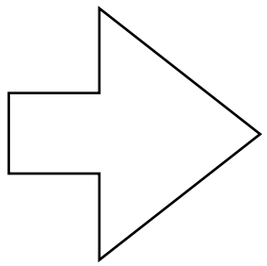
Structural Patterns

Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns



Abstract Factory

Singleton

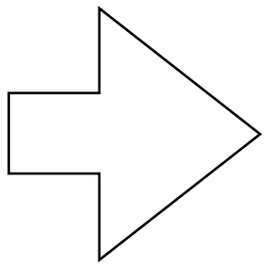
Factory Method

Prototype

Builder

Pattern types

Creational Patterns



Abstract Factory

Singleton

Factory Method

Prototype

Builder

Behavioural Patterns

Structural Patterns

The *abstract factory pattern* provides a way to encapsulate a group of individual factories with a common theme without specifying their concrete classes.

Let's see an example >>

If you want to create cars of ***different models*** from the ***same brand***

you need ***Mercedes Factory***

If you want ***another brand*** with ***different models***

You need additional ***Audi Factory***

Abstract Factory

Two factories have the same available public API for:

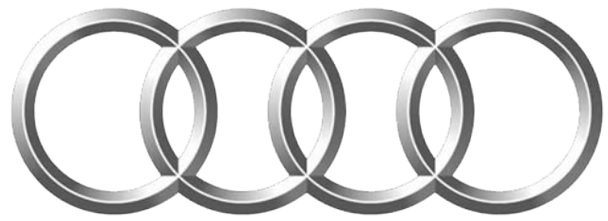
Creating a new car

Delivering it to customer

Developing new models

some other…

Mercedes Factory

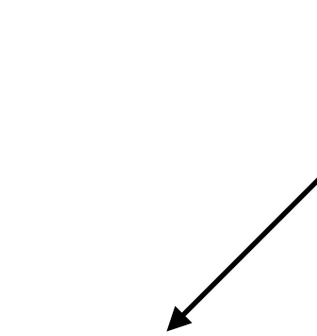


Audi Factory

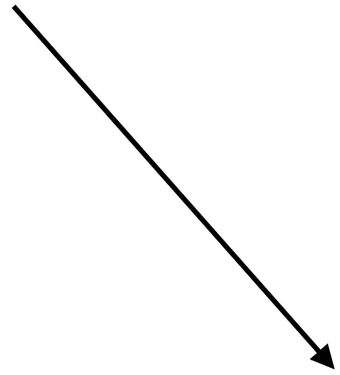
Abstract Factory

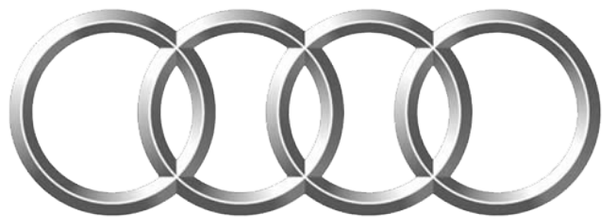
API can be extracted and implemented as an Interface

CarFactory <<Interface>>



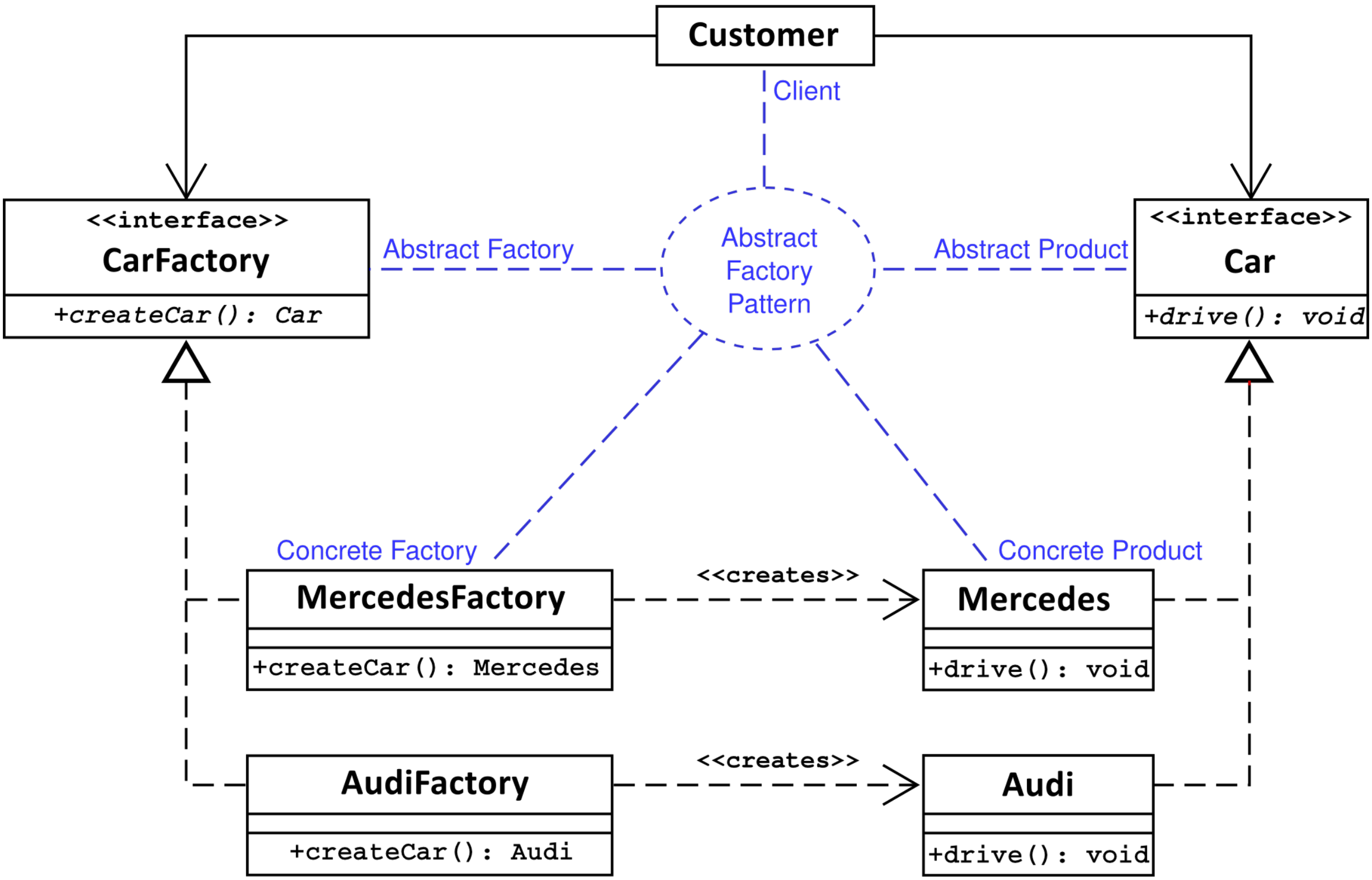
Mercedes Factory





Audi Factory

Abstract Factory

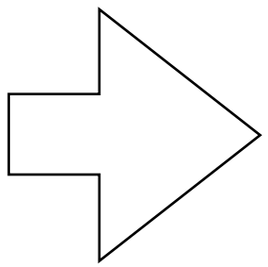


Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns



Abstract Factory

Singleton

Factory Method

Prototype

Builder

```java
public class Game {
    private final String name;
    private final Player player;
    private final Level level;
    private final Board board;
    private final Renderer renderer;

    public Game(String name, Player player, Level level, Board board, Renderer renderer) {
        this.name = name;
        this.player = player;
        this.level = level;
        this.board = board;
        this.renderer = renderer;
    }

    public Game(String name, Player player, Level level, Board board) {
        this(name, player, level, board, new Renderer());
    }

    public Game(String name, Player player, Level level) {
        this(name, player, level, new Board());
    }

    public Game(String name, Player player) {
        this(name, player, new Level());
    }

    public Game(String name) {
        this(name, new Player());
    }

    public Game() {
        this("Default game");
    }
}
```

The ***telescoping constructor anti-pattern*** occurs when the increase of object constructor parameter combination leads to an exponential list of constructors

The intent of *the Builder design pattern* is to separate the construction of a complex object from its representation.

```
public class Game {
   private final Player player;
   private final Level level;

   public Game(Player player, Level level) {
      this.player = player;
      this.level = level;
   }
}
```

Static builder class

```
public class Game {
  private final Player player;
  private final Level level;

  public Game(Player player, Level level) {
    this.player = player;
    this.level = level;
  }

  public static class Builder {

  }
}
```

Static builder class

```
public class Game {
   private final Player player;
   private final Level level;

   public Game(Player player, Level level) {
      this.player = player;
      this.level = level;
   }

   public static class Builder {
      private Player player;
      private Level level;

      public Game build() {
         return new Game(player, level);
      }
   }
}
```

Static builder class

```
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }

    public static class Builder {
        private Player player;
        private Level level;

        public Builder setPlayer(Player player) {
            this.player = player;
            return this;
        }
        public Builder setLevel(Level level) {
            this.level = level;
            return this;
        }
        public Game build() {
            return new Game(player, level);
        }
    }
}
```

Static builder class

```
public class Game {
    private final Player player;
    private final Level level;

    public Game(Player player, Level level) {
        this.player = player;
        this.level = level;
    }

    public static class Builder {
        private Player player;
        private Level level;

        public Builder setPlayer(Player player) {
            this.player = player;
            return this;
        }
        public Builder setLevel(Level level) {
            this.level = level;
            return this;
        }
        public Game build() {
            return new Game(player, level);
        }
    }
}
```

Duplication

Inner builder class

```
public class Game {
   private final Player player;
   private final Level level;

   private Game() {

   }
}
```

Inner builder class

```
public class Game {
   private Player player;
   private Level level;

   private Game() {}

   public static Builder builder() {
       return new Game().new Builder();
   }

   public class Builder {

   }
}
```

Inner builder class

```
public class Game {
    private Player player;
    private Level level;
    private Game() {}

    public static Builder builder() {
         return new Game().new Builder();
     }

    public class Builder {
        private Builder() {}

        public Builder setPlayer(Player player) {
            Game.this.player = player;
            return this;
        }

        public Builder setLevel(Level level) {
            Game.this.level = level;
            return this;
        }

        public Game build() {
            return Game.this;
        }
    }
}
```

Inner builder class

```
public class Game {
    private Player player;
    private Level level;
    private Game() {}

    public static Builder builder() {
         return new Game().new Builder();
     }

    public class Builder {
        private Builder() {}

        public Builder setPlayer(Player player) {
            Game.this.player = player;
            return this;
        }

        public Bu
            Game.t
            return
        }

        public Game build() {
            return Game.this;
        }
    }
}
```

Does not create new object on each build() call

Inner builder class + Cloneable

```
public class Game implements Cloneable {

   private Game() {}

   public Game clone() {
      Game game;
      try {
         game = (Game) super.clone();
         // clone mutable instance fields if needed
      } catch (CloneNotSupportedException e) {
         e.printStackTrace();
         throw new RuntimeException();
      }
      return game;
   }
}
```

Inner builder class + Cloneable

Before

```
public Game build() {
   return Game.this;
}
```

After

```
public Game build() {
   return Game.this.clone();
}
```

Usage:

```
public static void main(String[] args) {
   Game game = Game.builder()
         .setLevel(new Level())
         .setPlayer(new Player())
         .build();
}
```

VS.

```
public static void main(String[] args) {
   Game game = new Game(new Player(), new Level());
}
```

VS.

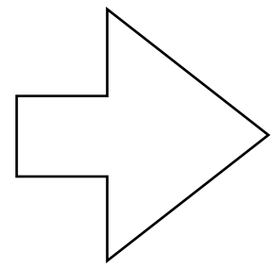
```
public static void main(String[] args) {
   Game game = new Game();
   game.setPlayer(new Player());
   game.setLevel(new Level());
}
```

Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns



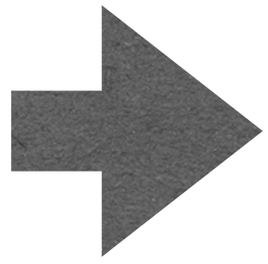
Chain of responsibility
Command
Interpreter
Iterator
Mediator
Momento
Observer
State
Strategy
Template Method
Visitor

Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns



Chain of responsibility

Command

Interpreter

Iterator

Mediator

Momento

Observer

State

Strategy

Template Method

Visitor

Chain of responsibility

The chain-of-responsibility is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism also exists for adding new processing objects to the end of this chain.

Chain of responsibility

The idea is to *process the message by yourself or to redirect it to someone else.*

Let's see an example >>

Chain of responsibility

You need to repair a car

Chain of responsibility

Please, repair my car

Chain of responsibility

Can I repair it?

No

Can I repair it?

Please, repair my car

Chain of responsibility

Can I repair it?

No

Can I repair it?

Not everything

Can I repair it?

Please, repair my car

Chain of responsibility

Can I repair it?

No

Can I repair it?

Not everything

Can I repair it?

Almost

Can I repair it?

Please, repair my car

Chain of responsibility

Can I repair it?

No

Can I repair it?

Not everything

Can I repair it?

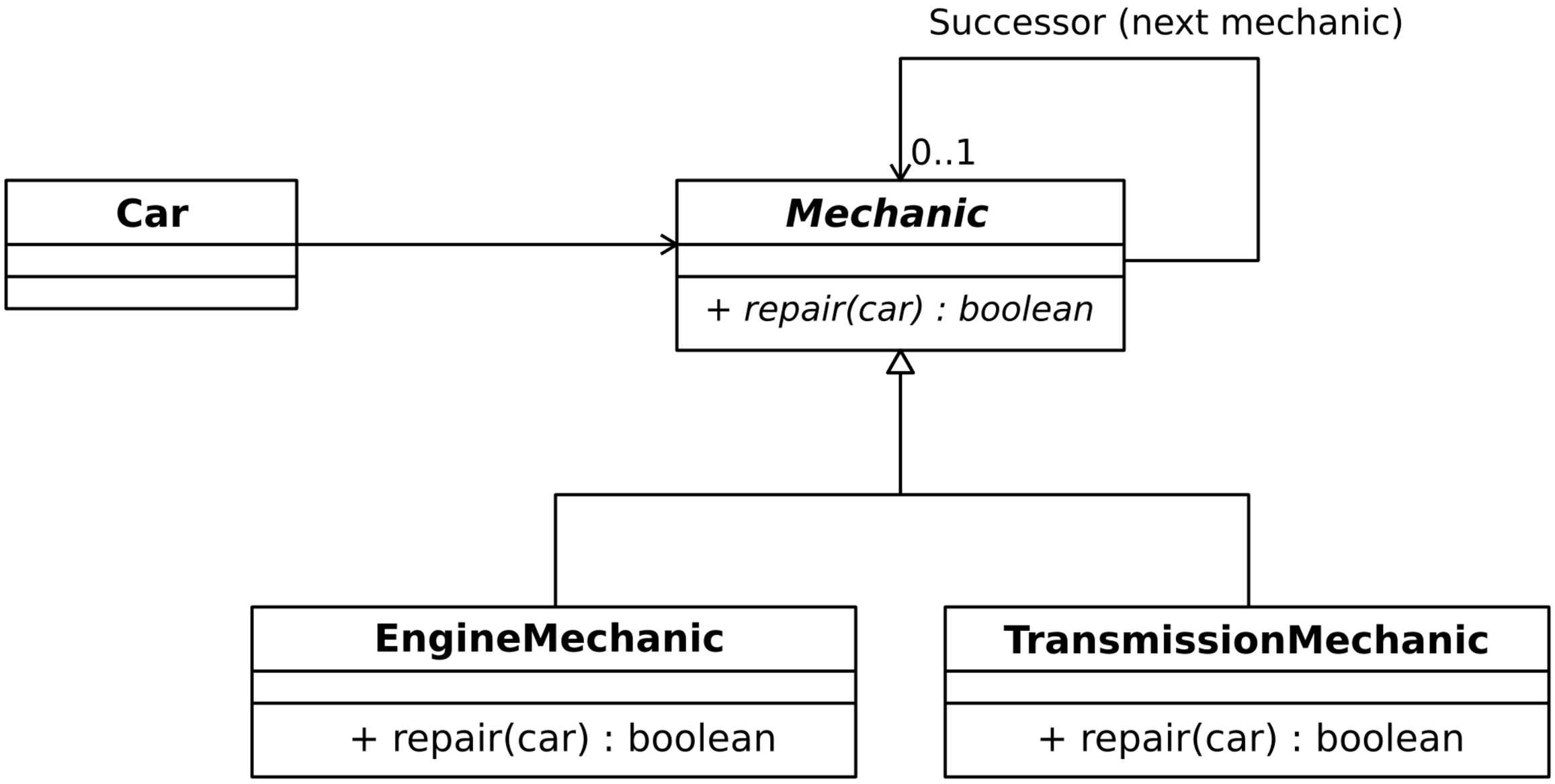
Almost

Can I repair it?

Done

Please, repair my car

Chain of responsibility

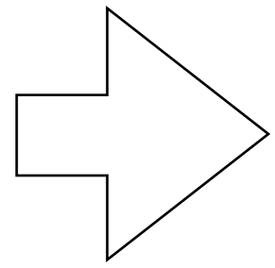


Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns



Chain of responsibility

Command

Interpreter

Iterator

Mediator

Momento

Observer

State

Strategy

Template Method

Visitor

The visitor pattern provides an ability to add new operations to existing object structures without modifying those structures

Let's see an example >>

Visitor

Help Darth Vader to check the dislocation of his forces.

Visitor

!!!

1. **Death Star *accepts* Darth Vader.**
2. **Darth Vader *visits* Death Star.**

accept

visit

Visitor

Troopers on Death Star suggest Darth Vader what to *<u>visit next:</u>* Star Destroyer.

accept

visit

accept

visit

Visitor

In the end he visits troopers.

accept

visit

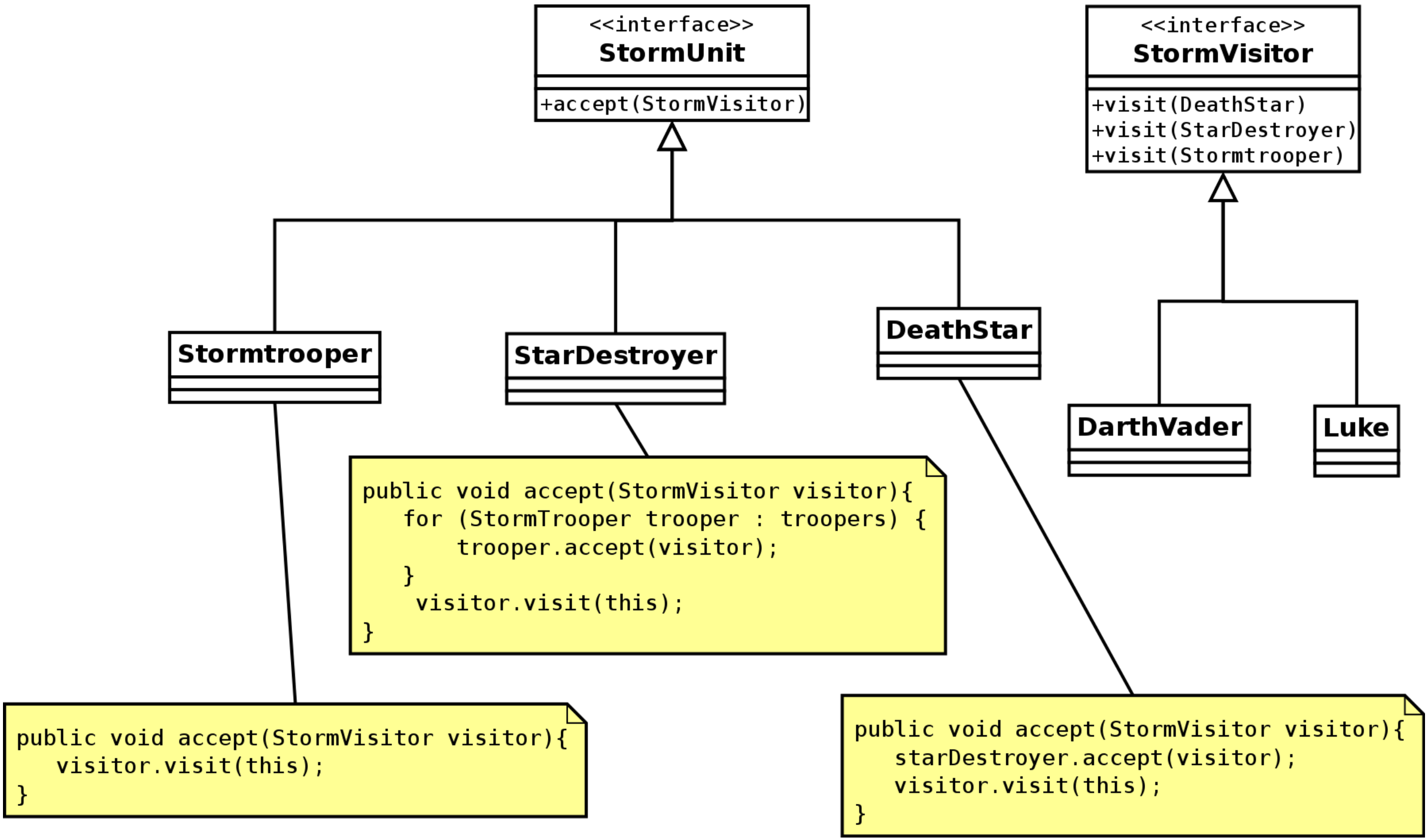
accept

visit

accept

visit

Visitor

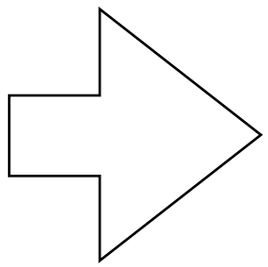

<<interface>>
StormUnit
+accept(StormVisitor)
<<interface>>
StormVisitor
+visit(DeathStar)
+visit(StarDestroyer)
+visit(Stormtrooper)
Stormtrooper
StarDestroyer
DeathStar
DarthVader
Luke
public void accept(StormVisitor visitor){
for (StormTrooper trooper : troopers) {
trooper.accept(visitor);
}
visitor.visit(this);
}
public void accept(StormVisitor visitor){
visitor.visit(this);
}
public void accept(StormVisitor visitor){
starDestroyer.accept(visitor);
visitor.visit(this);
}

Pattern types

Creational Patterns

Behavioural Patterns

Structural Patterns



Adapter

Bridge

Composite

Decorator

Facade

Flyweight

Proxy

The composite pattern lets a client to treat a group or a single instance uniformly - to have the same interface.

Let's see an example >>

Composite

Darth Vader wants to control one trooper or group of troopers

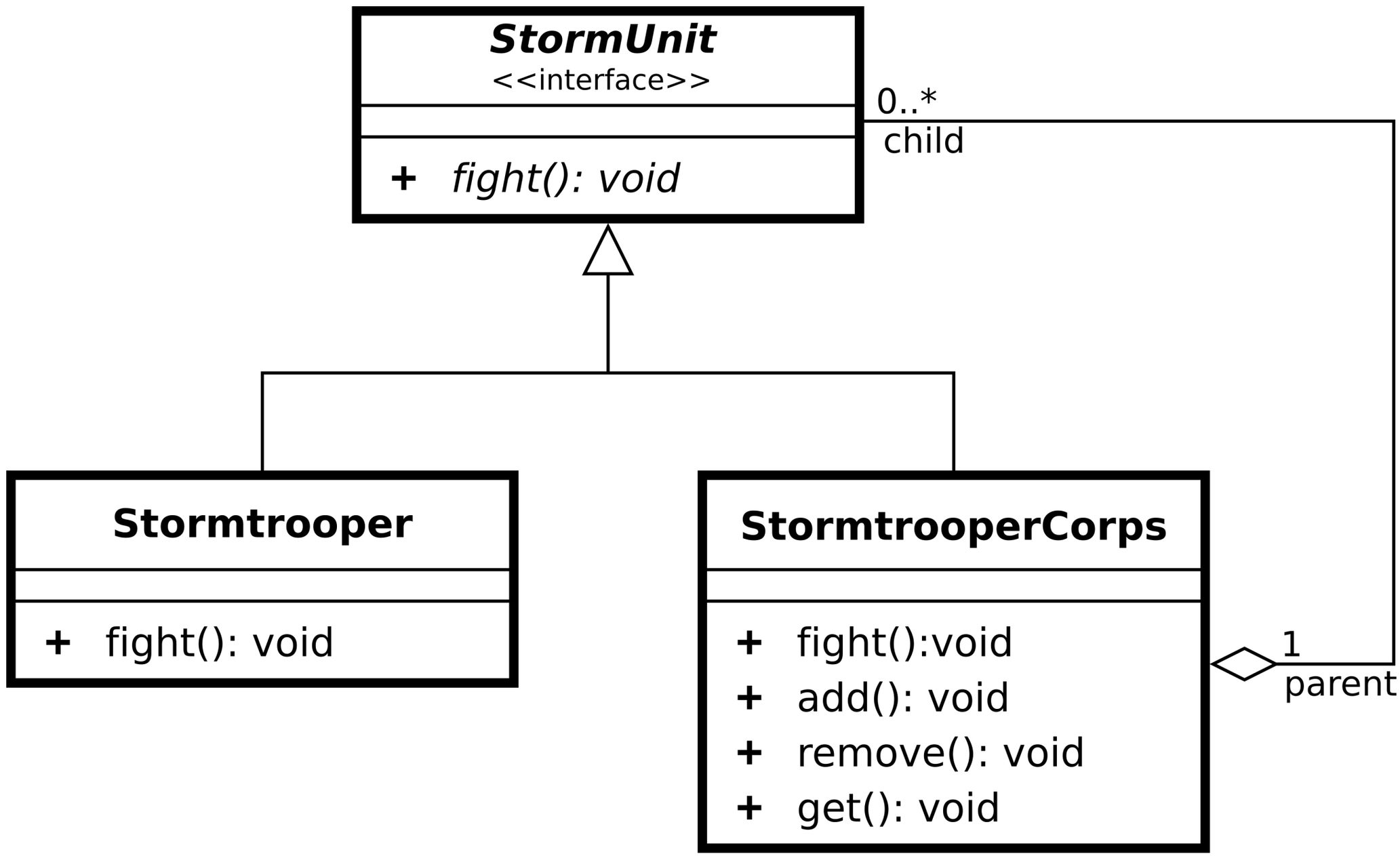
Composite

… or even groups of groups of troopers

Composite

Darth Vader doesn't care how many troopers to control - one or many

Composite


StormUnit
<<interface>>
+ fight(): void
0..*
child
Stormtrooper
+ fight(): void
StormtrooperCorps
+ fight():void
+ add(): void
+ remove(): void
+ get(): void
1
parent

The End.