Yield и async-await

как оно все устроено внутри и как этим воспользоваться

Дашкевич Иван
spaceorc@skbkontur.ru
СКБ Контур

План

- ICFP Contest
- Задача 2018 года
- Архитектура решения
- Yield-ы, async-и, task-like типы
- Решение задачи
- Идеи применения

ICFP Contest

- 72 часа
- Одна задача
- Любой язык программирования
- Команда любого размера
- Все время разные организаторы
- Разнообразные задачи

ICFP Contest

First prize

[Language 1] is the programming tool of choice for discriminating hackers.

Second prize

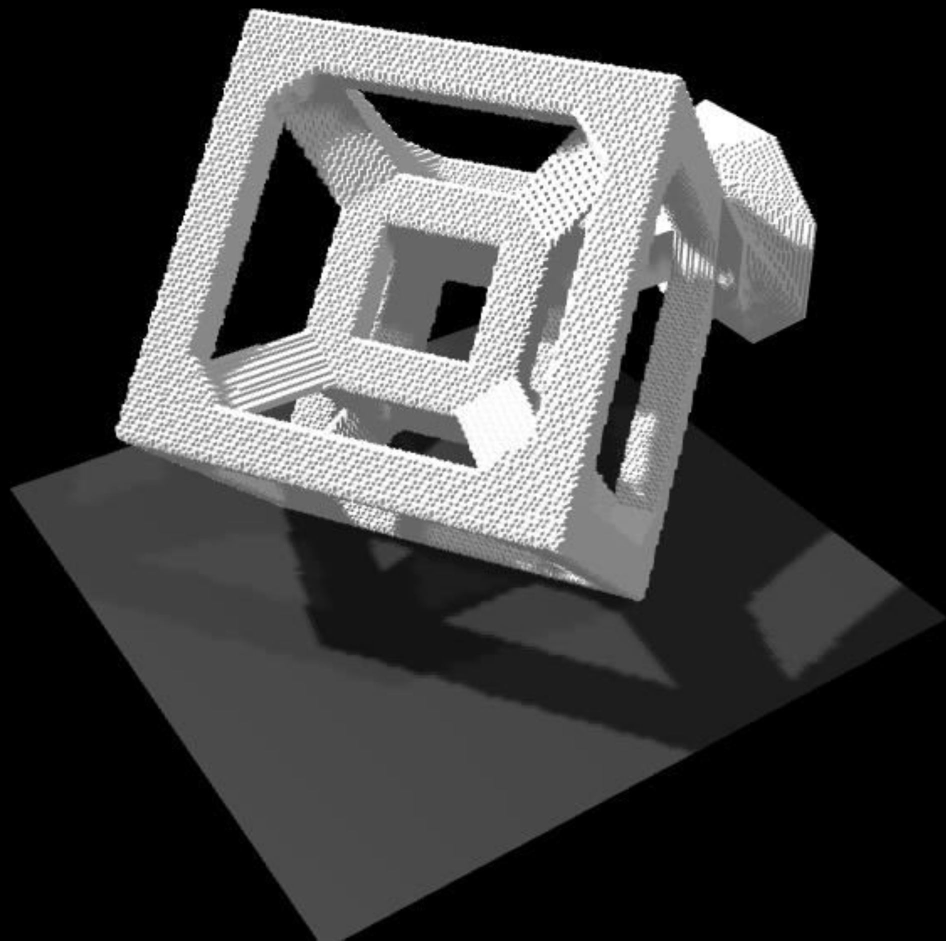
[Language 2] is a fine programming tool for many applications.

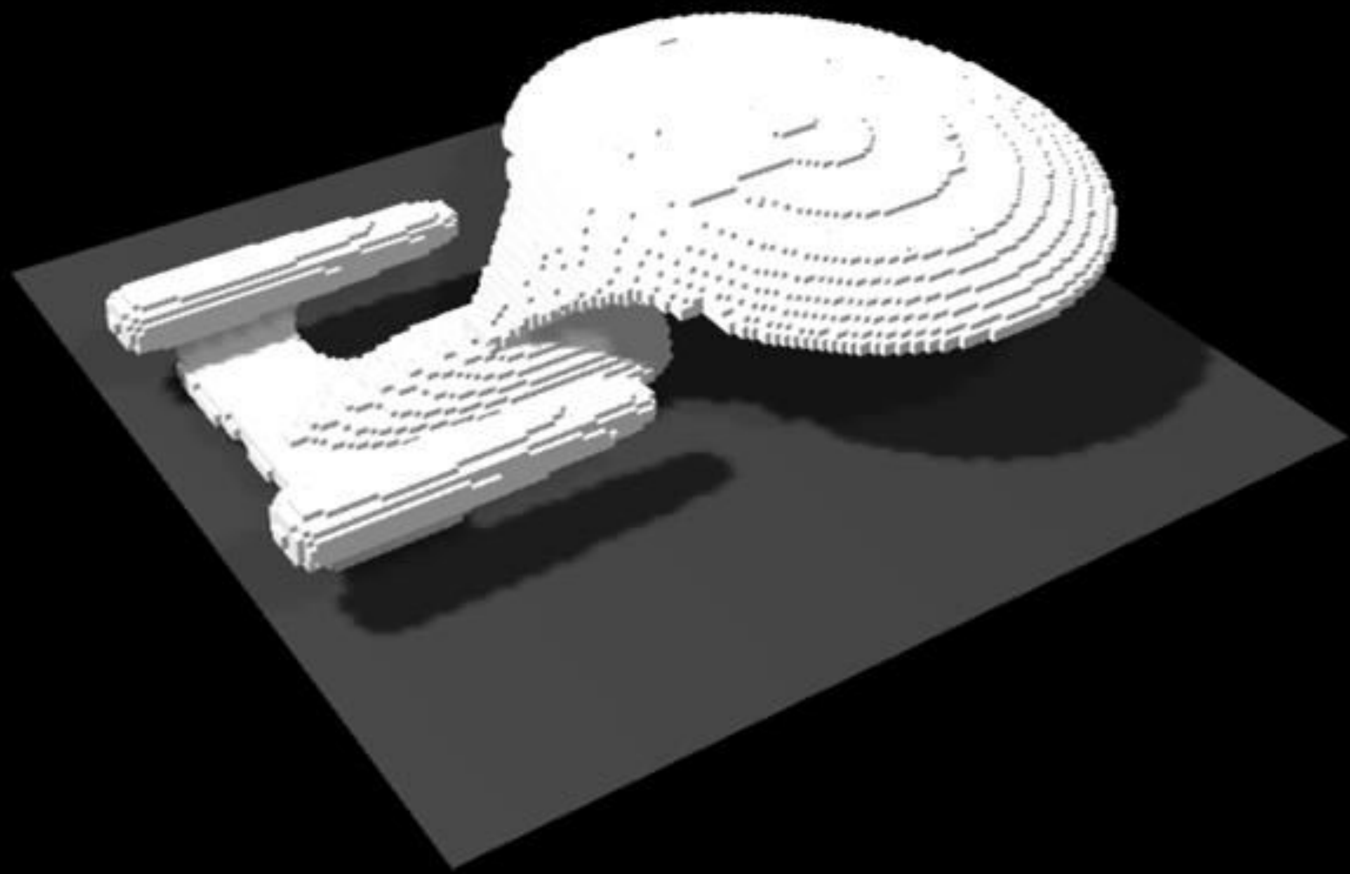
Third prize

[Language 3] is also not too shabby.

ICFP Contest 2018

- Нано-боты.
- 3D-печать в объеме от 20*20*20 до 250*250*250





ICFP Contest 2018

- Боты могут:

ICFP Contest 2018

- Боты могут:
    - Перемещаться прыжками

ICFP Contest 2018

- Боты могут:
  - Перемещаться прыжками
  - Размножаться делением и сливаться обратно

ICFP Contest 2018

- Боты могут:
  - Перемещаться прыжками
  - Размножаться делением и сливаться обратно
  - Заполнять материей и очищать клеточку рядом с собой

ICFP Contest 2018

- Боты могут:
  - Перемещаться прыжками
  - Размножаться делением и сливаться обратно
  - Заполнять материей и очищать клеточку рядом с собой
  - Действуя в группе, заполнять и очищать сразу целую область

ICFP Contest 2018

- Боты могут:
  - Перемещаться прыжками
  - Размножаться делением и сливаться обратно
  - Заполнять материей и очищать клеточку рядом с собой
  - Действуя в группе, заполнять и очищать сразу целую область
  - Пропускать ход

ICFP Contest 2018

- Боты могут:
  - Перемещаться прыжками
  - Размножаться делением и сливаться обратно
  - Заполнять материей и очищать клеточку рядом с собой
  - Действуя в группе, заполнять и очищать сразу целую область
  - Пропускать ход
- Минимизировать затраты энергии

ICFP Contest 2018

- Боты могут:
    - Перемещаться прыжками
    - Размножаться делением и сливаться обратно
    - Заполнять материей и очищать клеточку рядом с собой
    - Действуя в группе, заполнять и очищать сразу целую область
    - Пропускать ход
- Минимизировать затраты энергии
- Материя не может “висеть в воздухе”

ICFP Contest 2018

- Боты могут:
  - Перемещаться прыжками
  - Размножаться делением и сливаться обратно
  - Заполнять материей и очищать клеточку рядом с собой
  - Действуя в группе, заполнять и очищать сразу целую область
  - Пропускать ход
- Минимизировать затраты энергии
- Материя не может “висеть в воздухе”
- Решение - цепочка команд ботам

Как решить любую проблему

"Любую проблему можно решить путём введения дополнительного уровня абстракции, кроме проблемы слишком большого количества уровней абстракции"

(c) кто-то в интернете

```
interface IStrategy {



}
```

```
interface IStrategy {
  void Tick(); // вызываем каждый ход



}
```

```
interface IStrategy {
  void Tick(); // вызываем каждый ход

  // InProgress, Failed, Done
  StrategyStatus Status { get; }
}
```

```
interface IStrategy {
  void Tick(); // вызываем каждый ход

  // InProgress, Failed, Done
  StrategyStatus Status { get; }
}

bool IsFinished(this IStrategy s) =>
  s.Status != StrategyStatus.InProgress;
```

Стратегии-кирпичики

Strategy

Strategy

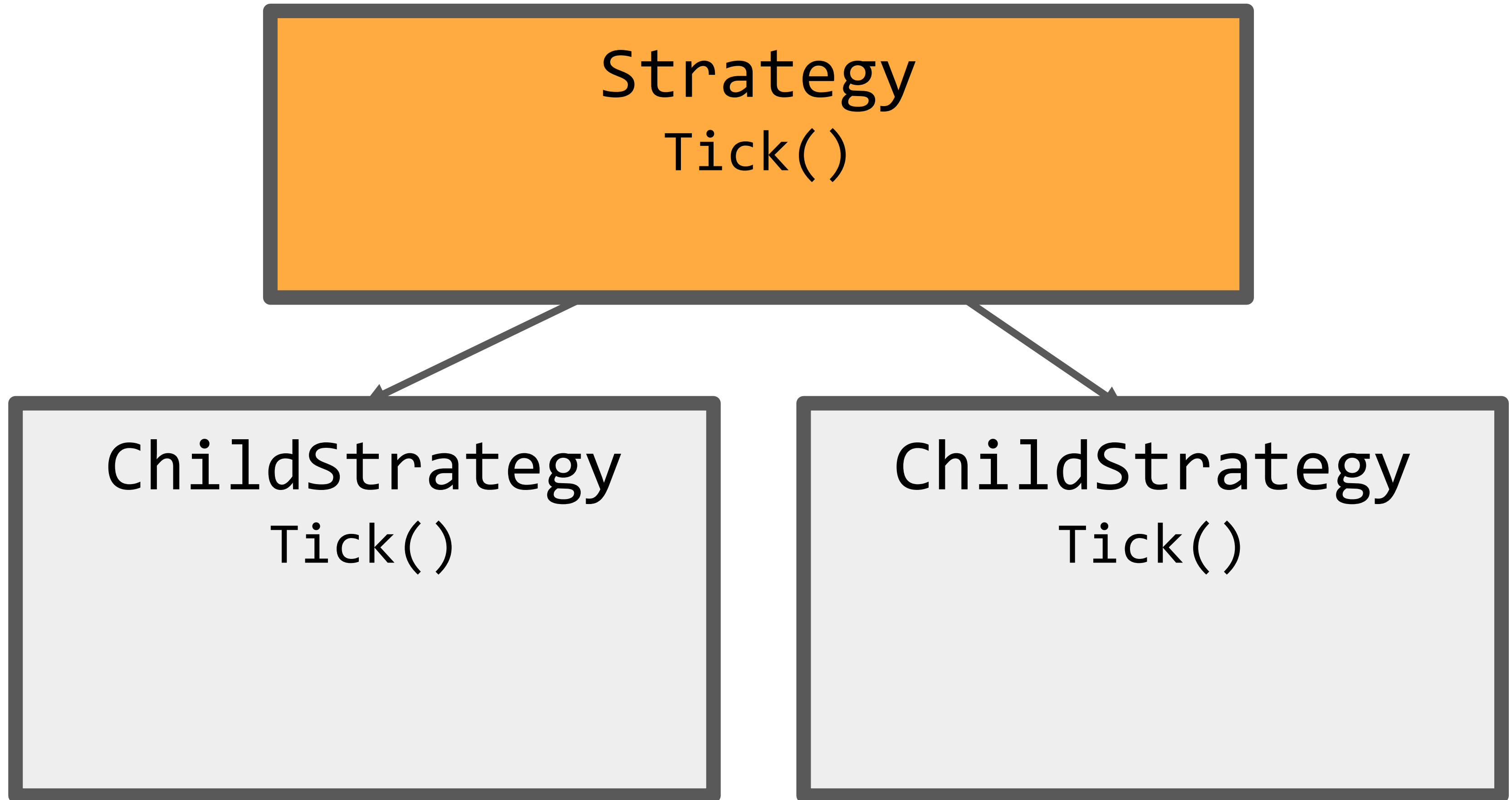
ChildStrategy

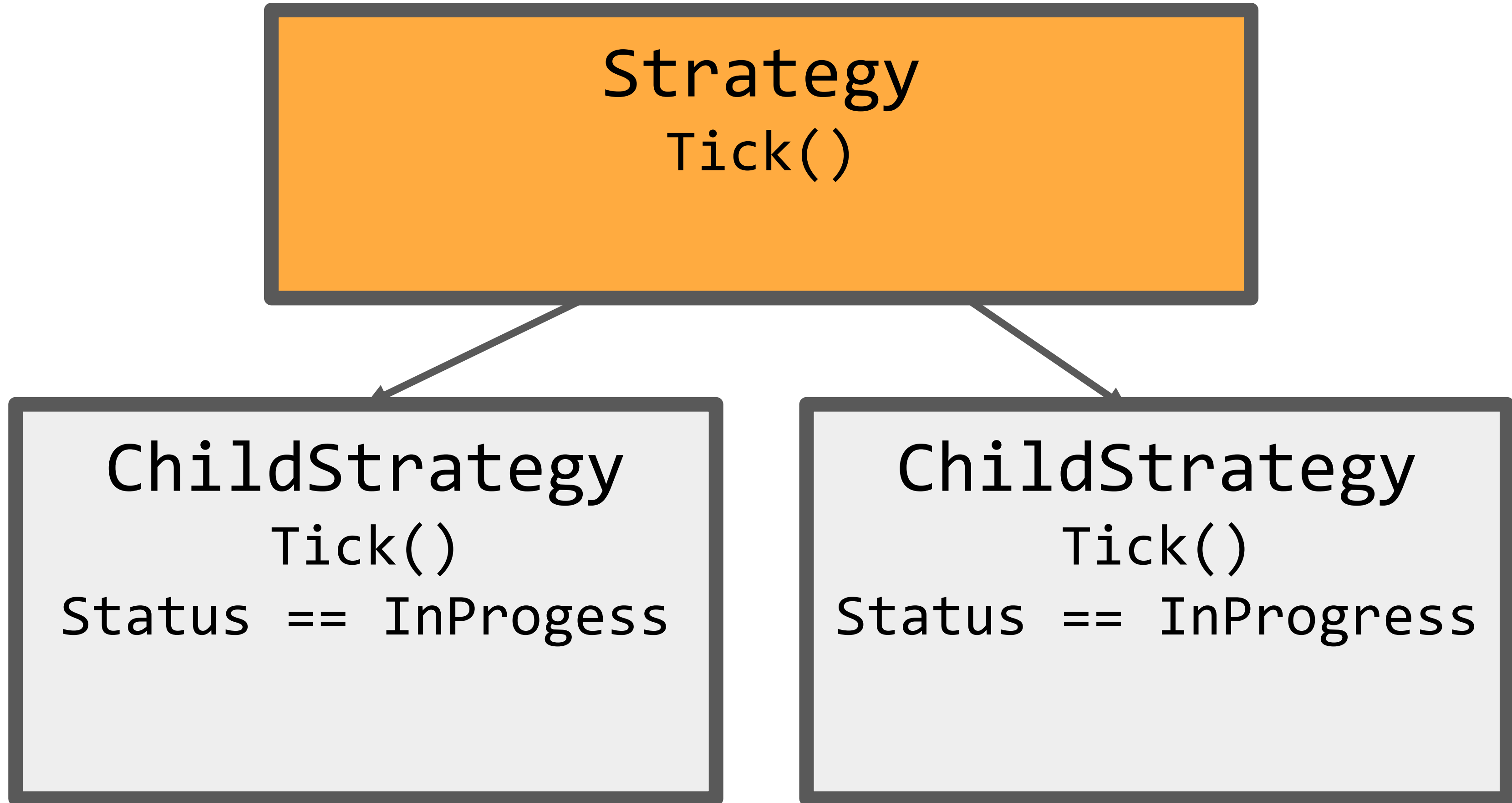
ChildStrategy

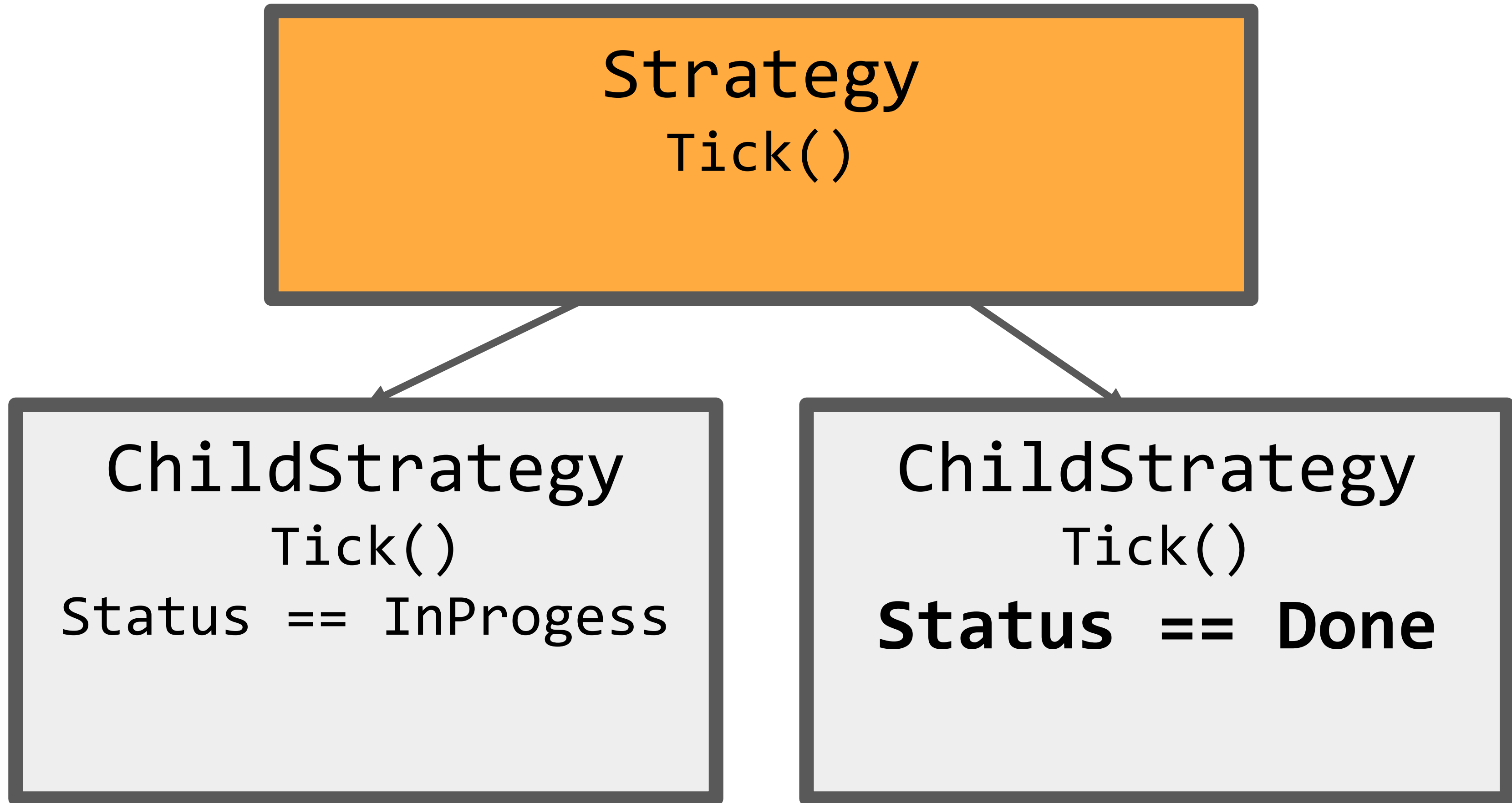
Strategy
Tick()

ChildStrategy

ChildStrategy


Strategy
Tick()
ChildStrategy
Tick()
ChildStrategy
Tick()


Strategy
Tick()
ChildStrategy
Tick()
Status == InProgess
ChildStrategy
Tick()
Status == InProgress


Strategy
Tick()
ChildStrategy
Tick()
Status == InProgess
ChildStrategy
Tick()
Status == Done

Strategy
Tick()

ChildStrategy
Tick()
Status == InProgess

Strategy
Tick()

ChildStrategy
Tick()
Status == Done

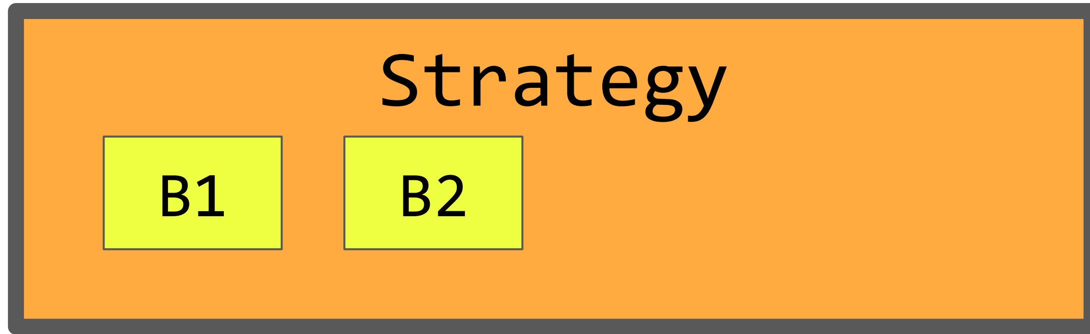
Strategy

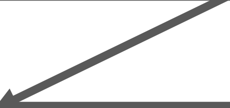
Tick()

Strategy

B1

B2





ChildStrategy

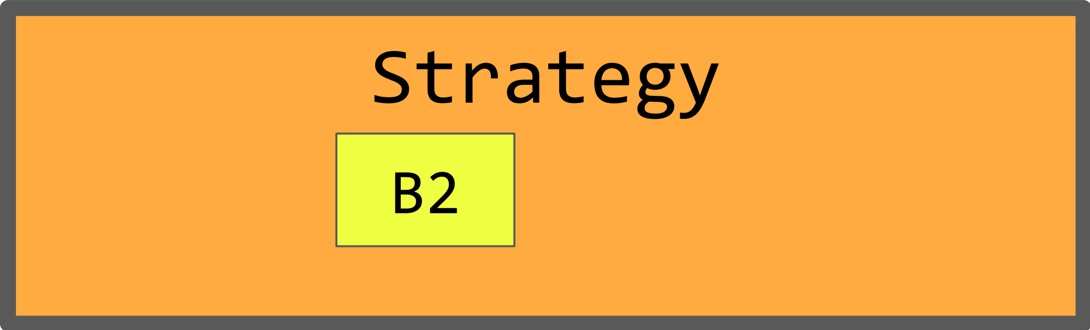
```
new ChildStrategy(B1);
```

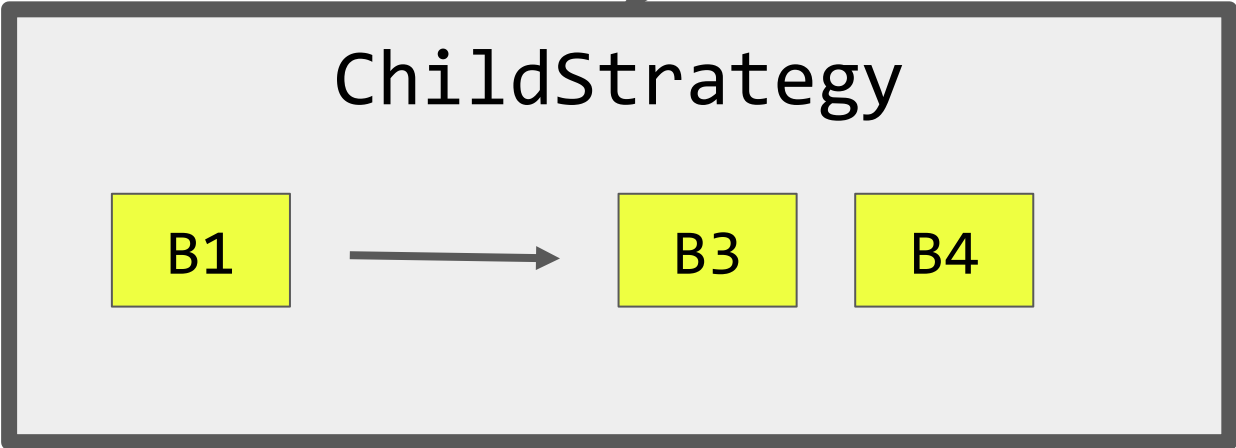
Strategy

B2

ChildStrategy

B1


Strategy
B2


ChildStrategy
B1
B3
B4

Strategy

B2

ChildStrategy

B3

B4

Strategy

B2

ChildStrategy

Status == Done

B3

B4

Strategy

B2 B3 B4

ChildStrategy

Status == Done

Strategy

B2 B3 B4

Перемещение бота в требуемую точку

```
class Goto : IStrategy {
  void Tick() {









  }
}
```

```
class Goto : IStrategy {
  void Tick() {

    var commands = TryFindPath();

















  }
}
```

```
class Goto : IStrategy {
  void Tick() {

    var commands = TryFindPath();
    if (commands == null) {
      Status = StrategyStatus.Failed;
      return;
    }








  }
}
```

```
class Goto : IStrategy {
  void Tick() {

    var commands = TryFindPath();
    if (commands == null) {
      Status = StrategyStatus.Failed;
      return;
    }

    if (commands.Count == 0) {
      Status = StrategyStatus.Done;
      return;
    }


  }
}
```

```
class Goto : IStrategy {
  void Tick() {

    var commands = TryFindPath();
    if (commands == null) {
      Status = StrategyStatus.Failed;
      return;
    }

    if (commands.Count == 0) {
      Status = StrategyStatus.Done;
      return;
    }
    state.SetBotCommand(bot, commands[0]);

  }
}
```

```
class Goto : IStrategy {
  void Tick() {
    if (commands == null) {
      commands = TryFindPath();
      if (commands == null) {
        Status = StrategyStatus.Failed;
        return;
      }
    }
    if (commands.Count == 0) {
      Status = StrategyStatus.Done;
      return;
    }
    state.SetBotCommand(bot, commands[0]);
    commands.RemoveAt(0);
  }
}
```

```
class Goto : IStrategy {
  void Tick() {
    if (IsBadPath(commands)) {
      commands = TryFindPath();
      if (commands == null) {
        Status = StrategyStatus.Failed;
        return;
      }
    }
    if (commands.Count == 0) {
      Status = StrategyStatus.Done;
      return;
    }
    state.SetBotCommand(bot, commands[0]);
    commands.RemoveAt(0);
  }
}
```

Более сложный кирпичик: стратегия завершения работы

{

}

```
{
  // все боты бегут к точке (0,0,0), пока могут
}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)
}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)

  while (/* есть еще другие боты, кроме master */) {




  }

}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)

  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут



  }


}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут
  // master = бот, стоящий в точке (0,0,0)
  while (/* есть еще другие боты, кроме master */) {
    // другие боты бегут к точке (1,0,0), пока могут
    // slave = бот, стоящий в точке (1,0,0)

  }

}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)

  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master
  }


}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)

  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master
  }

  // остался только один - master - завершаем работу машины
}
```

Yield return

```
IEnumerable<string> GetValues() {
  yield return "lalala";
  yield return "bubububu";
}
```

```
IEnumerable<string> GetValues() {
  return new GetValues_StateMachine(state: -2);
}
```

```
IEnumerable<string> GetValues() {
  return new GetValues_StateMachine(state: -2);
}

class GetValues_StateMachine : IEnumerable<string>,
                               IEnumerator<string> {
  int state;

  IEnumerator<string> GetEnumerator() {
    state = 0;
    return this;
  }



}
```

```
IEnumerable<string> GetValues() {
  return new GetValues_StateMachine(state: -2);
}

class GetValues_StateMachine : IEnumerable<string>,
                               IEnumerator<string> {
  int state;

  IEnumerator<string> GetEnumerator() {
    state = 0;
    return this;
  }

  // IEnumerator implementation
  string Current { get; private set; }
  bool MoveNext() { ... }
}
```

```
bool MoveNext() {

}
```

```
IEnumerable<string> GetValues() {
  yield return "lalala";
  yield return "bububu";
}
```

```
bool MoveNext() {
  switch (state) {









  }
}
```

```
IEnumerable<string> GetValues() {
  yield return "lalala";
  yield return "bububu";
}
```

```
bool MoveNext() {
  switch (state) {
    case 0:
      Current = "lalala";
      state = 1;
      return true;








  }
}
```

```
IEnumerable<string> GetValues() {
  yield return "lalala";
  yield return "bububu";
}
```

```
bool MoveNext() {
  switch (state) {
    case 0:
      Current = "lalala";
      state = 1;
      return true;

    case 1:
      Current = "bububu";
      state = 2;
      return true;




  }
}
```

```
IEnumerable<string> GetValues() {
  yield return "lalala";
  yield return "bububu";
}
```

```
bool MoveNext() {
  switch (state) {
    case 0:
      Current = "lalala";
      state = 1;
      return true;

    case 1:
      Current = "bububu";
      state = 2;
      return true;

    case 2:
      state = -1;
      return false;
  }
}
```

```
IEnumerable<string> GetValues() {
  yield return "lalala";
  yield return "bububu";
}
```

```
{
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)


  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  // все боты бегут к точке (0,0,0), пока могут

  // master = бот, стоящий в точке (0,0,0)


  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0)));

  // master = бот, стоящий в точке (0,0,0)


  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0)));

  var master = bots.Single(bot => bot.Position == (0,0,0));


  while (/* есть еще другие боты, кроме master */) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0)));

  var master = bots.Single(bot => bot.Position == (0,0,0));

  var others = bots.Except(new[] {master}).ToList();
  while (others.Any()) {

    // другие боты бегут к точке (1,0,0), пока могут

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0))));

  var master = bots.Single(bot => bot.Position == (0,0,0));

  var others = bots.Except(new[] {master}).ToList();
  while (others.Any()) {

    yield return others.Select(bot => new Goto(bot, (1,0,0)));

    // slave = бот, стоящий в точке (1,0,0)

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0)));

  var master = bots.Single(bot => bot.Position == (0,0,0));

  var others = bots.Except(new[] {master}).ToList();
  while (others.Any()) {

    yield return others.Select(bot => new Goto(bot, (1,0,0)));

    var slave = bots.Single(bot => bot.Position == (1,0,0));

    // вливаем slave в master

  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0)));

  var master = bots.Single(bot => bot.Position == (0,0,0));

  var others = bots.Except(new[] {master}).ToList();
  while (others.Any()) {

    yield return others.Select(bot => new Goto(bot, (1,0,0)));

    var slave = bots.Single(bot => bot.Position == (1,0,0));

    yield return new Merge(master, slave);
    others.Remove(slave);
  }
  // остался только один - master - завершаем работу машины
}
```

```
IEnumerable Run() {
  yield return bots.Select(bot => new Goto(bot, (0,0,0)));

  var master = bots.Single(bot => bot.Position == (0,0,0));

  var others = bots.Except(new[] {master}).ToList();
  while (others.Any()) {

    yield return others.Select(bot => new Goto(bot, (1,0,0)));

    var slave = bots.Single(bot => bot.Position == (1,0,0));

    yield return new Merge(master, slave);
    others.Remove(slave);
  }
  yield return new Halt(master);
}
```

IStrategy.Tick()

```
void Tick() {

}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();
















}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();









    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }

}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();


    if (enumerator.Current is IEnumerable<IStrategy> children) {




    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }

}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();


    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();




    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }

}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();


    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());


    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }

}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();


    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }

}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();

  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

“It's feasible but it is a bad idea. Iterator blocks were created to help you write custom iterators for collections, not for solving the general-purpose problem of implementing state machines.”

Eric Lippert https://stackoverflow.com/questions/1194853/implementing-a-state-machine-using-the-yield-keyword

Async/await

Async/await

- Специально разработан для нашего случая

Async/await

- Специально разработан для нашего случая
- Есть бонус:

Async/await

- Специально разработан для нашего случая
- Есть бонус:

```
var a = await something;        // так можно
var a = yield return something; // а так нет!
```

```
{
  var goto1 = new Goto(bot1, position1);
  var goto2 = new Goto(bot2, position2);










}
```

```
{
  var goto1 = new Goto(bot1, position1);
  var goto2 = new Goto(bot2, position2);

  yield return new[] { goto1, goto2 };








}
```

```
{
  var goto1 = new Goto(bot1, position1);
  var goto2 = new Goto(bot2, position2);

  yield return new[] { goto1, goto2 };

  if (goto1.Status == StrategyStatus.Done &&
      goto2.Status == StrategyStatus.Done)
  {
    // оба бота точно пришли к своим целям
  }
}
```

```
{
  var goto1 = new Goto(bot1, position1);
  var goto2 = new Goto(bot2, position2);

  if (await new [] { goto1, goto2 })
  {
    // оба бота точно пришли к своим целям
  }
}
```

```
{
  if (await new []
            {
              new Goto(bot1, position1),
              new Goto(bot2, position2)
            })
  {
    // оба бота точно пришли к своим целям
  }
}
```

Еще более сложный кирпичик: строим целый регион

```
class AssembleRegion : IStrategy {









}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;















}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;

  async StrategyTask<bool> Run () {









  }
}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;

  async StrategyTask<bool> Run () {

    var vertices = region.GetVertices();








  }
}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;

  async StrategyTask<bool> Run () {

    var vertices = region.GetVertices();

    var targets = vertices.Select(v => GetFreeNeighbor(v));





  }
}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;

  async StrategyTask<bool> Run () {

    var vertices = region.GetVertices();

    var targets = vertices.Select(v => GetFreeNeighbor(v));

    await targets.Select((t, i) => new Goto(bots[i], t));



  }
}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;

  async StrategyTask<bool> Run () {

    var vertices = region.GetVertices();

    var targets = vertices.Select(v => GetFreeNeighbor(v));

    if (!await targets.Select((t, i) => new Goto(bots[i], t)))
      return false;


  }
}
```

```
class AssembleRegion : IStrategy {
  Bot[] bots;
  Region region;

  async StrategyTask<bool> Run () {

    var vertices = region.GetVertices();

    var targets = vertices.Select(v => GetFreeNeighbor(v));

    if (!await targets.Select((t, i) => new Goto(bots[i], t)))
      return false;

    return await bots.Select(bot => new FillRegion(bot, region));
  }
}
```

StrategyTask<T> - аналог IEnumerator

StrategyTask<T> - аналог IEnumerator

- Возможность запускать продолжение работы
    - `enumerator.MoveNext()`

`StrategyTask<T>` - аналог `IEnumerator`

- Возможность запускать продолжение работы
  - `enumerator.MoveNext()`
- Информация о том, что метод завершен
  - `enumerator.MoveNext()` → **`bool`**

`StrategyTask<T>` - аналог `IEnumerator`

- Возможность запускать продолжение работы
  - `enumerator.MoveNext()`
- Информация о том, что метод завершен
  - `enumerator.MoveNext()` → **`bool`**
- Информация о том, чего метод ждет
  - `enumerator.Current`

StrategyTask<T> - аналог IEnumerator

- Возможность запускать продолжение работы
  - `enumerator.MoveNext()`
- Информация о том, что метод завершен
  - `enumerator.MoveNext()` → **`bool`**
- Информация о том, чего метод ждет
  - `enumerator.Current`
- Если завершен, то с каким результатом?

```
class StrategyTask<T> {

}
```

Возможность запускать продолжение работы

`enumerator.MoveNext()`

```
class StrategyTask<T> {

  void Continue();




}
```

Возможность запускать продолжение работы

```
enumerator.MoveNext()
```

```
class StrategyTask<T> {

  void Continue();

}
```

Информация о том, что метод завершен

`enumerator.MoveNext() → bool`

```
class StrategyTask<T> {

  void Continue();

  bool IsCompleted { get; }

}
```

Информация о том, что метод завершен

`enumerator.MoveNext() → bool`

```
class StrategyTask<T> {

  void Continue();

  bool IsCompleted { get; }



}
```

Информация о том, чего метод ждет

```
enumerator.Current
```

```
class StrategyTask<T> {

  void Continue();

  bool IsCompleted { get; }

  IStrategy[] Children { get; }


}
```

Информация о том, чего метод ждет

`enumerator.Current`

```
class StrategyTask<T> {

  void Continue();

  bool IsCompleted { get; }

  IStrategy[] Children { get; }


}
```

Если завершен, то с каким результатом?

```
class StrategyTask<T> {
  void Continue();
  bool IsCompleted { get; }
  IStrategy[] Children { get; }
  T Result { get; }
}
```

Если завершен, то с каким результатом?

IStragery.Tick() - теперь для **async/await**

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();

  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {
  if (enumerator == null)
    enumerator = Run().GetEnumerator();

  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {



  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }
    if (!enumerator.MoveNext()) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {




  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }
    if (                      ) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {
  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }





    if (                          ) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {
  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }

    if (task == null) task = Run();
    else task.Continue();

    if (task.IsCompleted      ) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {
  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }

    if (task == null) task = Run();
    else task.Continue();

    if (task.IsCompleted) {
      Status = StrategyStatus.Done;
      return;
    }
  }
}
```

```
void Tick() {
  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }

    if (task == null) task = Run();
    else task.Continue();

    if (task.IsCompleted) {
      Status = task.Result ? StrategyStatus.Done : StrtgyStatus.Failed;
      return;
    }
  }
}
```

```
void Tick() {
  while (true) {
    if (enumerator.Current is IEnumerable<IStrategy> children) {
      var pending = children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }

    if (task == null) task = Run();
    else task.Continue();

    if (task.IsCompleted) {
      Status = task.Result ? StrategyStatus.Done : StrtgyStatus.Failed;
      return;
    }
  }
}
```

```
void Tick() {
  while (true) {
    if (task != null) {
      var pending = task.Children.Where(s => !s.IsFinished()).ToList();
      pending.ForEach(s => s.Tick());
      if (pending.Any(s => !s.IsFinished()))
        return;
    }

    if (task == null) task = Run();
    else task.Continue();

    if (task.IsCompleted) {
      Status = task.Result ? StrategyStatus.Done : StrtgyStatus.Failed;
      return;
    }
  }
}
```

Что внутри async-метода

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
class Run_StateMachine : IAsyncStateMachine {

}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {






  }
}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;







    }







  }
}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }



  }
}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {



    }
  }
}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;


    }
  }
}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

```
async StrategyTask<bool> Run() {
  bool a = await new Goto(...);
  return a;
}
```

Awaiter

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

await expr ⇒ expr.GetAwaiter()

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetA
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

Обработка синхронного завершения

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

Awaiter : INotifyCompletion

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

Получение результата ожидания

```
class StrategyAwaiter {

  ...
}
```

```
class StrategyAwaiter {
  public StrategyAwaiter(IStrategy[] children) { ... }
  ...
}
```

```
class StrategyAwaiter {
  public StrategyAwaiter(IStrategy[] children) { ... }
  ...
}
```

```
await expr ⇒ expr.GetAwaiter()
```

```
static class StrategyExtensions {
  // await strategy → strategy.GetAwaiter()




  // await new[]{s1, s2} → new[]{s1, s2}.GetAwaiter()




}
```

```
class StrategyAwaiter {
  public StrategyAwaiter(IStrategy[] children) { ... }
  ...
}
```

`await expr ⇒ expr.GetAwaiter()`

```
static class StrategyExtensions {
  // await strategy → strategy.GetAwaiter()
  static StrategyAwaiter GetAwaiter(this IStrategy strategy)
  {
    return new StrategyAwaiter(new[] { strategy });
  }

  // await new[]{s1, s2} → new[]{s1, s2}.GetAwaiter()
  static StrategyAwaiter GetAwaiter(this IEnumerable<IStrtgy> strtgies)
  {
    return new StrategyAwaiter(strategies.ToArray());
  }
}
```

```
class StrategyAwaiter {
  public StrategyAwaiter(IStrategy[] children) {
    Children = children;
  }

  IStrategy[] Children { get; }










}
```

```
class StrategyAwaiter {
  public StrategyAwaiter(IStrategy[] children) {
    Children = children;
  }

  IStrategy[] Children { get; }

  bool IsCompleted => false;

}
```

Обработка синхронного завершения

```
class StrategyAwaiter {
  public StrategyAwaiter(IStrategy[] children) {
    Children = children;
  }

  IStrategy[] Children { get; }

  bool IsCompleted => false;

  bool GetResult() => Children.All(s => s.Status == StrtgyStatus.Done);

}
```

Получение результата ожидания

```
class StrategyAwaiter : INotifyCompletion {
  public StrategyAwaiter(IStrategy[] children) {
    Children = children;
  }

  IStrategy[] Children { get; }

  bool IsCompleted => false;

  bool GetResult() => Children.All(s => s.Status == StrtgyStatus.Done);

  // INotifyCompletion
  void OnCompleted(Action continuation) {
    this.continuation = continuation;
  }

}
```

Awaiter : INotifyCompletion

```
class StrategyAwaiter : INotifyCompletion {
  public StrategyAwaiter(IStrategy[] children) {
    Children = children;
  }

  IStrategy[] Children { get; }

  bool IsCompleted => false;

  bool GetResult() => Children.All(s => s.Status == StrtgyStatus.Done);

  // INotifyCompletion
  void OnCompleted(Action continuation) {
    this.continuation = continuation;
  }

  void Continue() => this.continuation();
}
```

TaskBuilder

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_St
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

Надо правильно выбрать тип taskBuilder-a

```
[AsyncMethodBuilder(typeof(StrategyTaskBuilder<>))]
class StrategyTask<T> {
  ...
}
```

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

Создается фабричным методом Create

```
class StrategyTaskBuilder<T> {

}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

}
```

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskB                          lder<bool>.Create();
  stateMachine.taskB                          e);
  return stateMachine.taskBuilder.Task;
}
```

Должен возвращать строящийся Task в соответствующем свойстве

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();










}
```

```
async StrategyTask<bool> Run() {
  var stateMachine = new Run_StateMachine(state: -1);
  stateMachine.taskBuilder = StrategyTaskBuilder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

```
async StrategyTask<bool> Run() {
  var stateMachine =                        ate: -1);
  stateMachine.taskB                        lder<bool>.Create();
  stateMachine.taskBuilder.Start(stateMachine);
  return stateMachine.taskBuilder.Task;
}
```

Должен запустить stateMachine в методе Start

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();









}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();









}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

Завершать Task в методе SetResult

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();
```

```
}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();

  void SetResult(T value) => Task.SetResult(value); // ???

}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

```
class Run_StateMachine : IAsyncStateMachine {
  void MoveNext() {
    if (state == -1) {
      awaiter = new Goto(...).GetAwaiter();
      state = 0;
      if (!awaiter.IsCompleted) {
        taskBuilder.AwaitOnCompleted(awaiter, this);
        return;
      }
    }
    if (state == 0) {
      a = awaiter.GetResult();
      state = -2;
      taskBuilder.SetResult(a);
      return;
    }
  }
}
```

Запускать ожидание awaiter-a в методе AwaitOnCompleted

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();

  void SetResult(T value) => Task.SetResult(value); // ???






}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();

  void SetResult(T value) => Task.SetResult(value); // ???

  void AwaitOnCompleted<TAwaiter, TSM>(
    ref TAwaiter awaiter, ref TSM stateMachine)
    where TAwaiter : INotifyCompletion
    where TSM : IAsyncStateMachine
  {


  }
}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();

  void SetResult(T value) => Task.SetResult(value); // ???

  void AwaitOnCompleted<TAwaiter, TSM>(
    ref TAwaiter awaiter, ref TSM stateMachine)
    where TAwaiter : INotifyCompletion
    where TSM : IAsyncStateMachine
  {
    awaiter.OnCompleted(() => stateMachine.MoveNext());

  }
}
```

```
class StrategyTaskBuilder<T> {
  static StrategyTaskBuilder<T> Create() => new StrtgyTaskBuilder<T>();

  StrategyTask<T> Task { get; } = new StrategyTask<T>();

  void Start<TSM>(ref TSM stateMachine) => stateMachine.MoveNext();

  void SetResult(T value) => Task.SetResult(value); // ???

  void AwaitOnCompleted<TAwaiter, TSM>(
    ref TAwaiter awaiter, ref TSM stateMachine)
    where TAwaiter : INotifyCompletion
    where TSM : IAsyncStateMachine
  {
    awaiter.OnCompleted(() => stateMachine.MoveNext());
    Task.SetAwaiter(awaiter); // ???
  }
}
```

StrategyTask<T>

```
class StrategyTask<T> {
  void Continue();
  bool IsCompleted { get; }
  IStrategy[] Children { get; }
  T Result { get; }
}
```

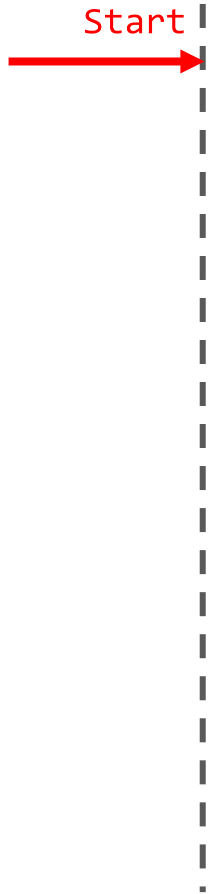
```
class StrategyTask<T> {
  void Continue();

  bool IsCompleted { get; }

  IStrategy[] Children { get; }

  T Result { get; }



}
```

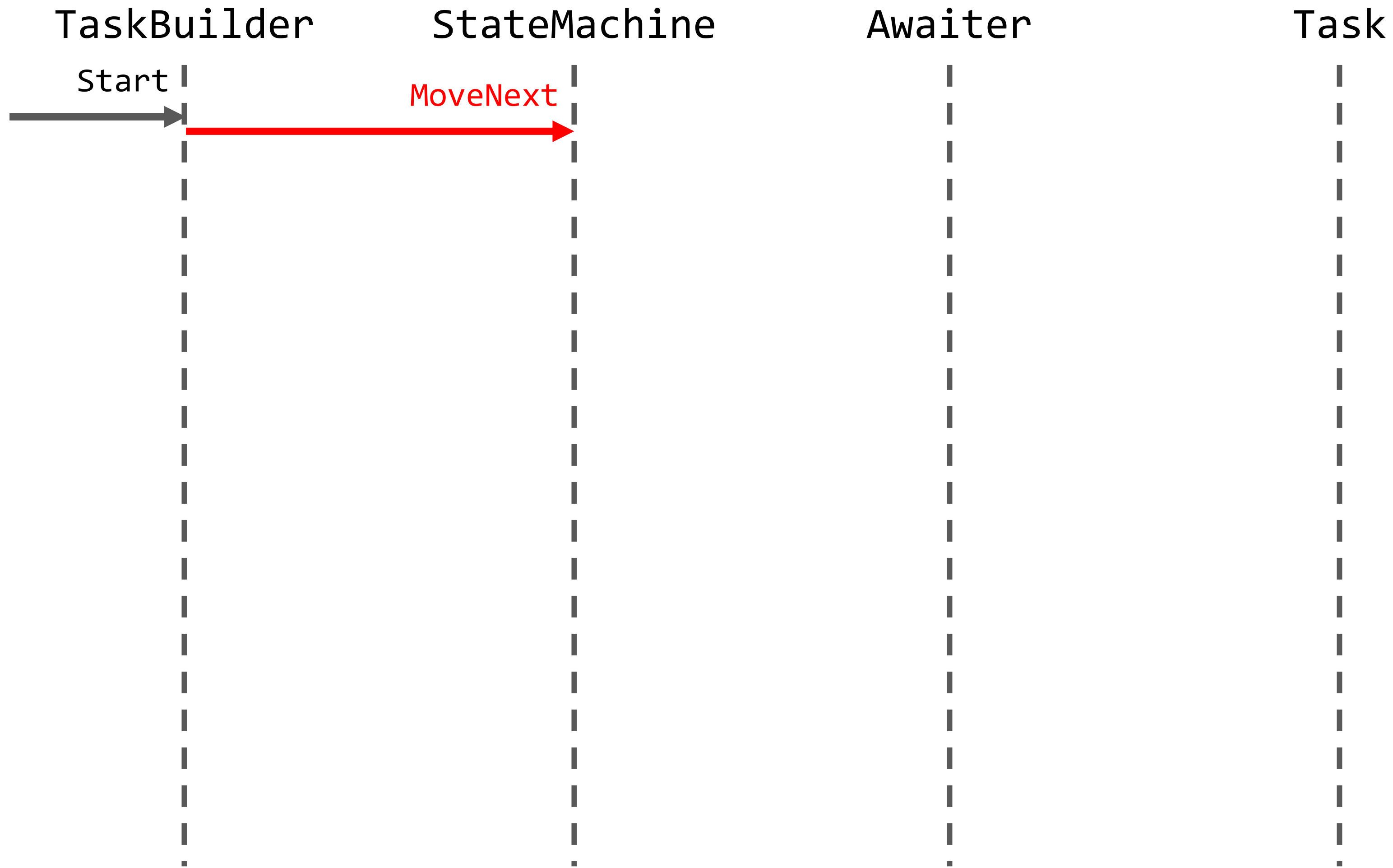
```
class StrategyTask<T> {
  void Continue();

  bool IsCompleted { get; private set; }

  IStrategy[] Children { get; }

  T Result { get; private set; }

  void SetResult(T value) {
    Result = value;
    IsCompleted = true;
  }



}
```

```
class StrategyTask<T> {
  void Continue() => this.awaiter.Continue();

  bool IsCompleted { get; private set; }

  IStrategy[] Children => this.awaiter.Children;

  T Result { get; private set; }

  void SetResult(T value) {
    Result = value;
    IsCompleted = true;
  }

  void SetAwaiter(StrategyTaskAwaiter awaiter) {
    this.awaiter = awaiter;
  }
}
```

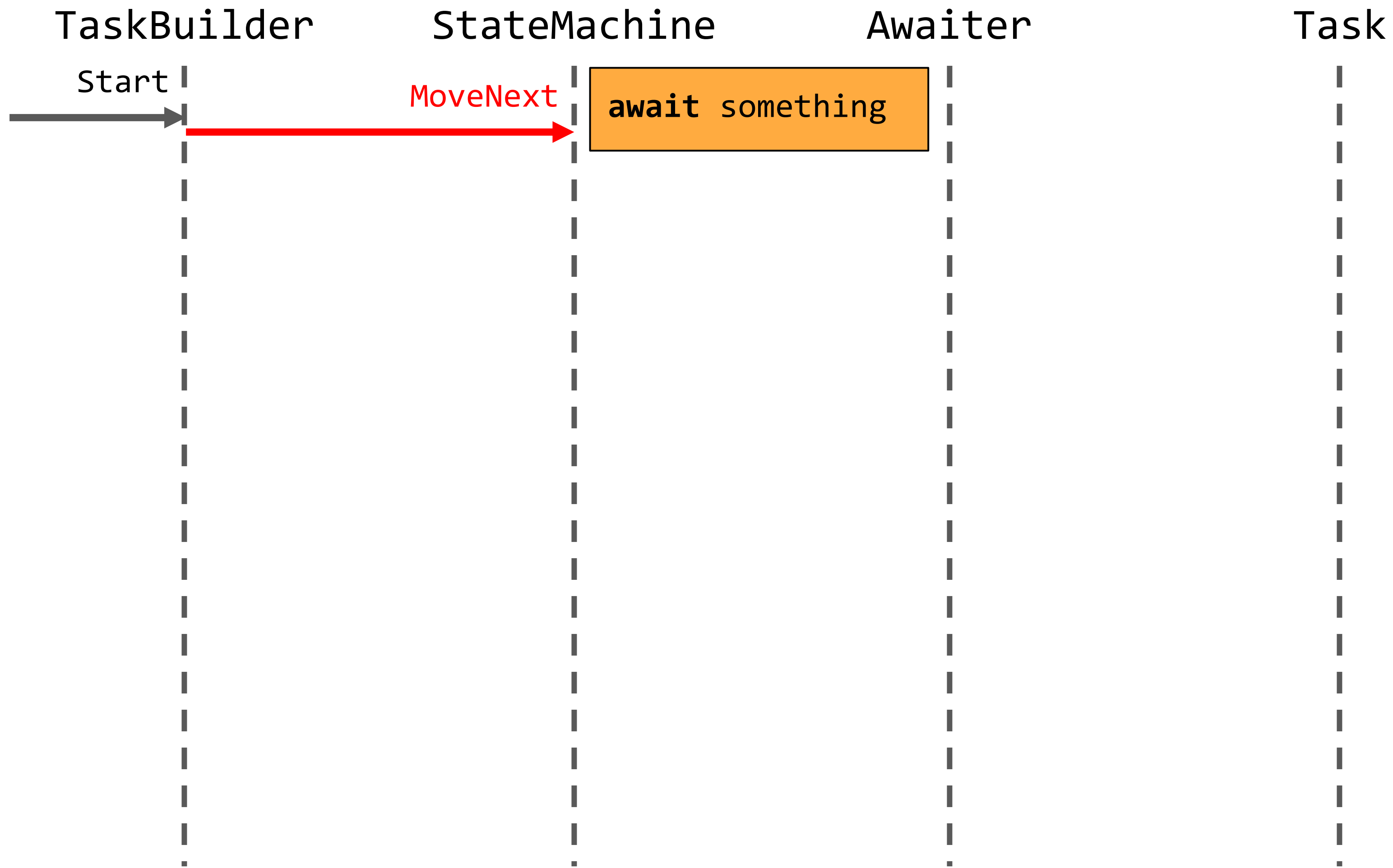
TaskBuilder StateMachine Awaiter Task

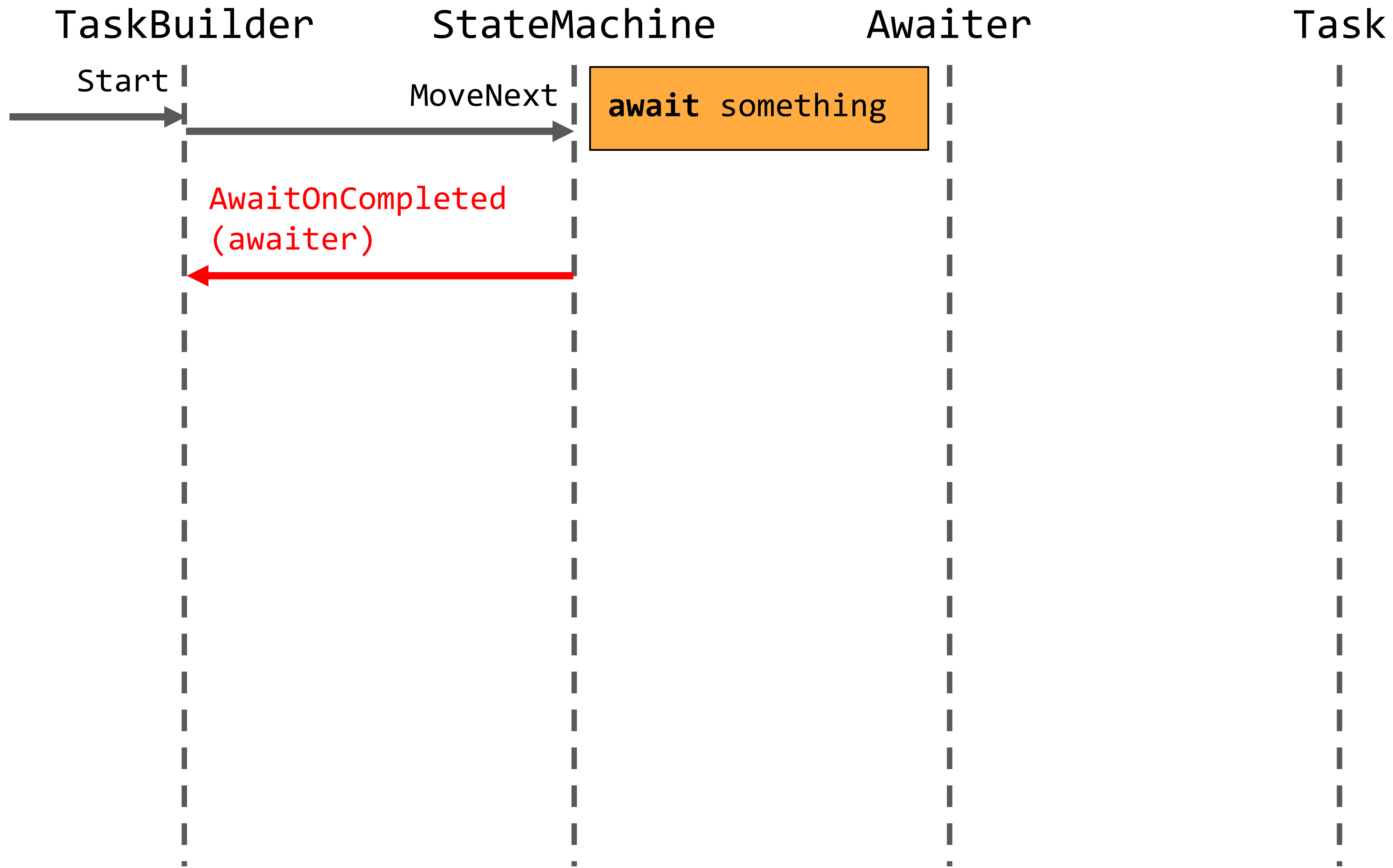
TaskBuilder StateMachine Awaiter Task

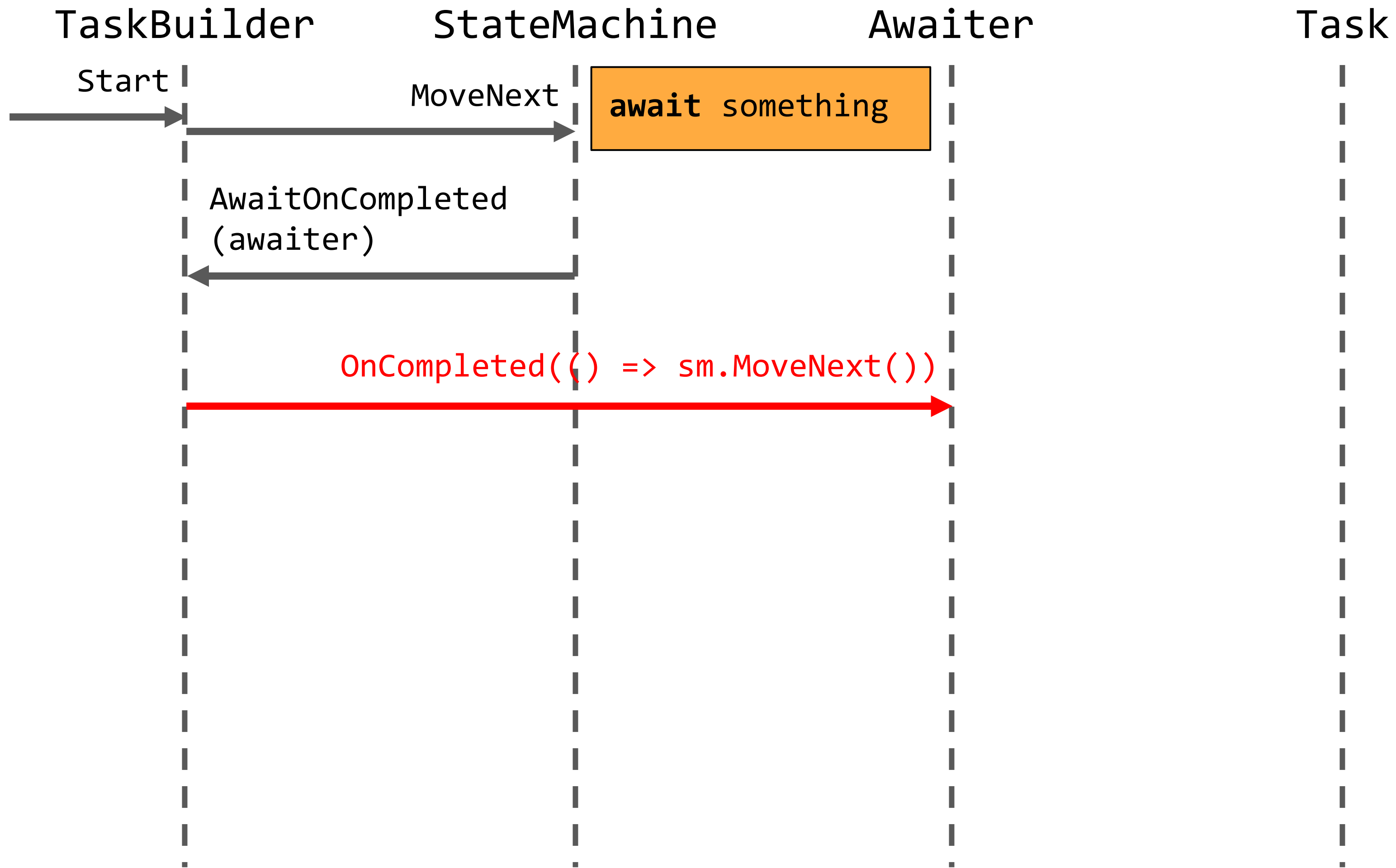


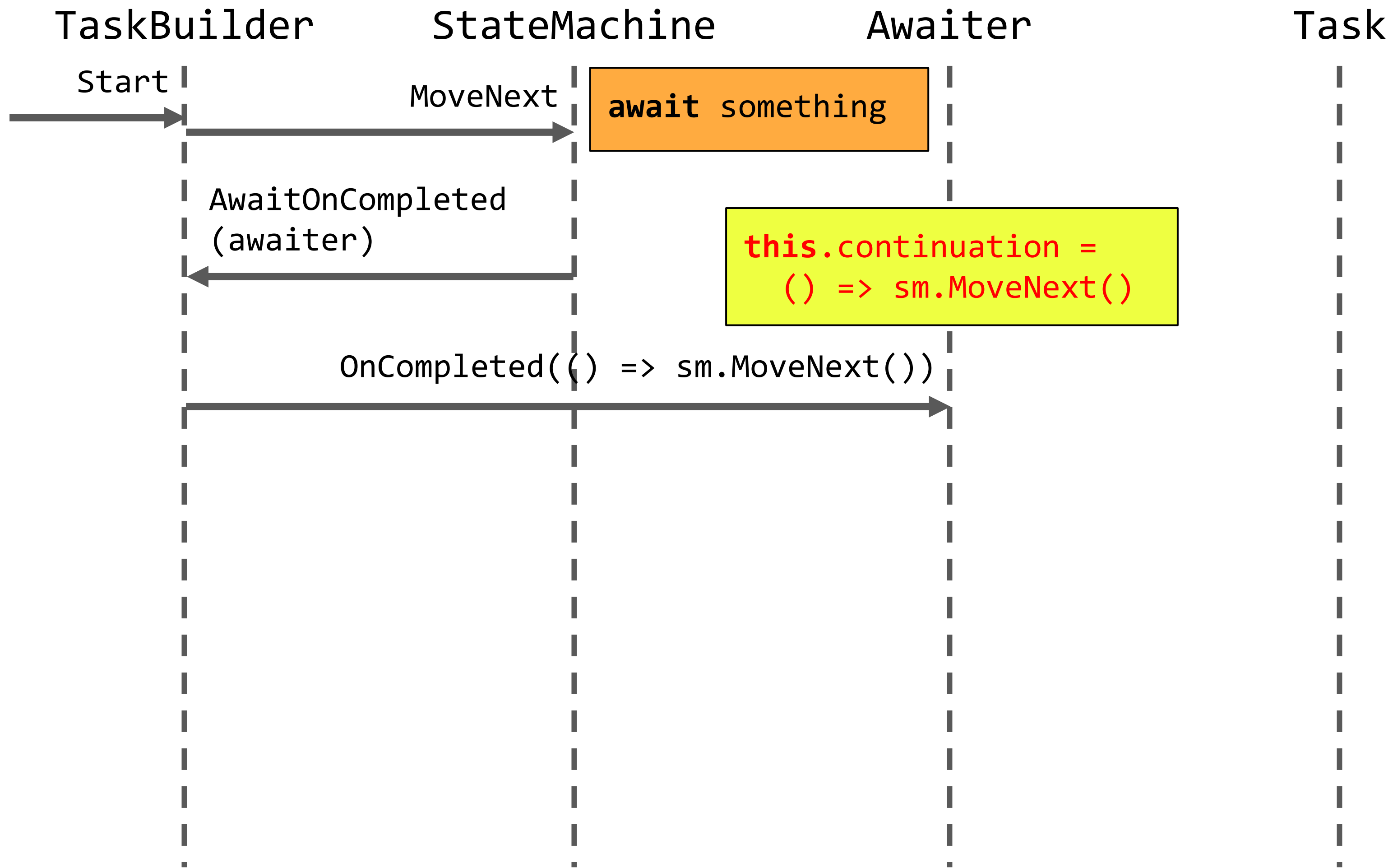


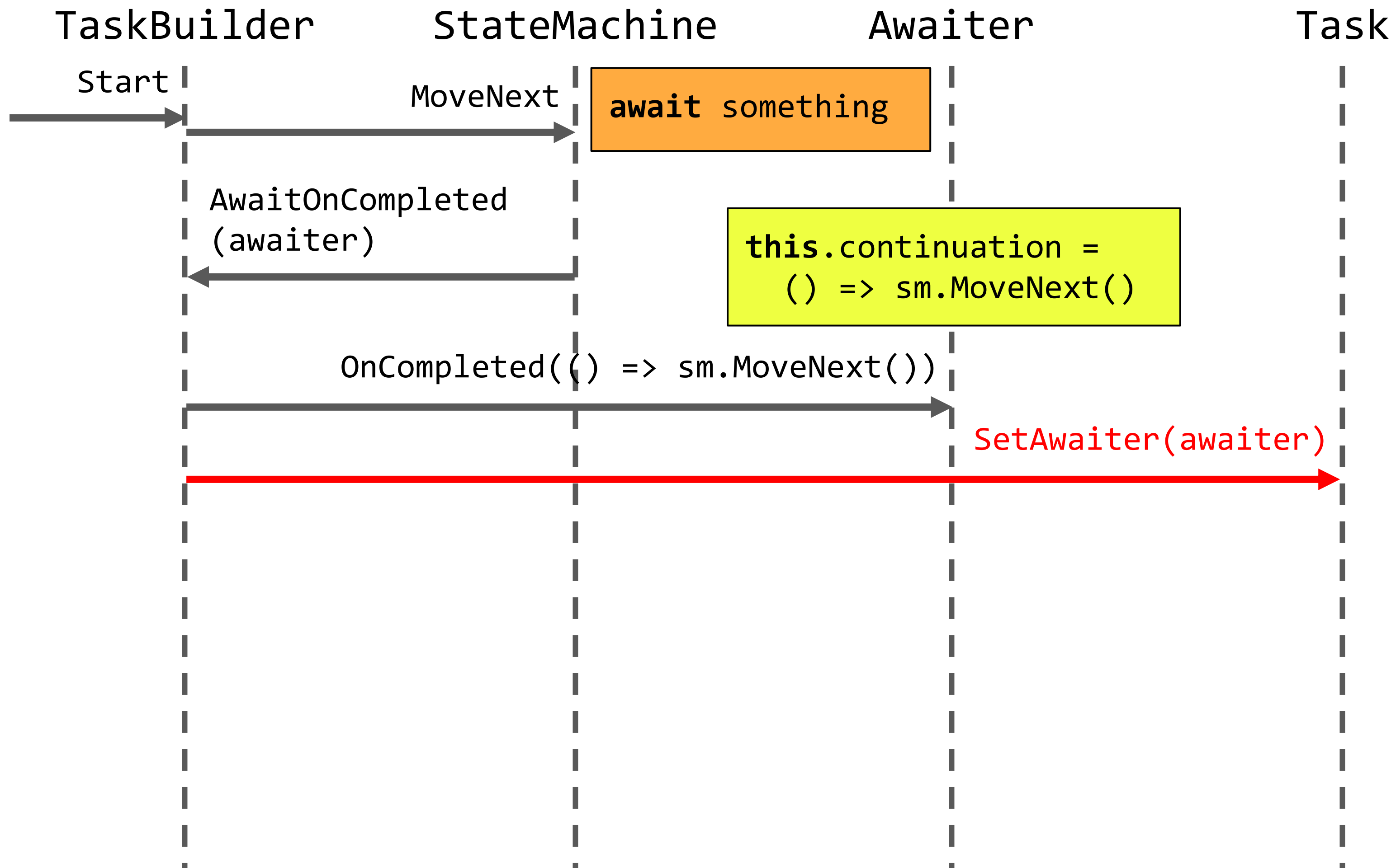
TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext

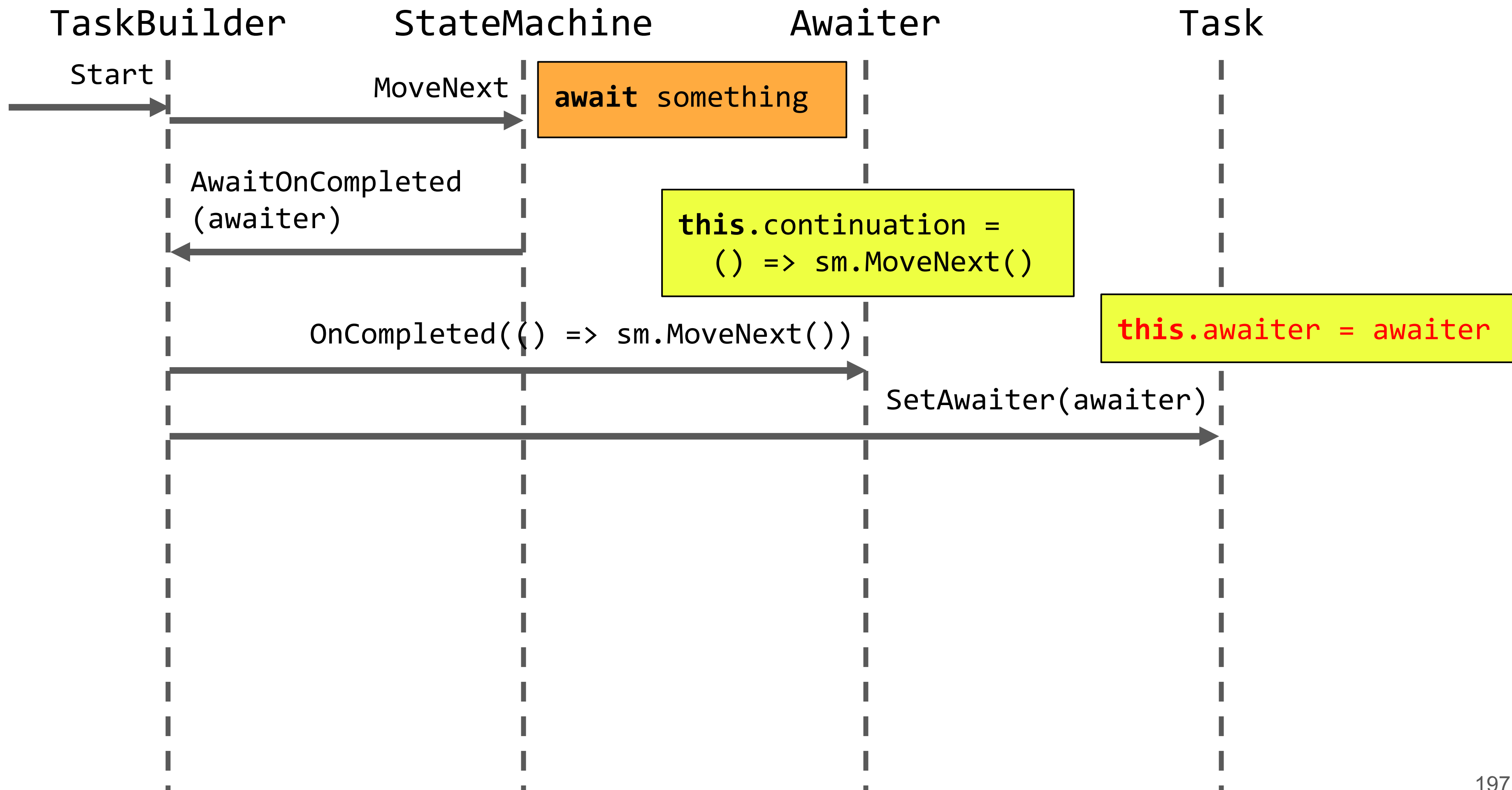

TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something


TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)

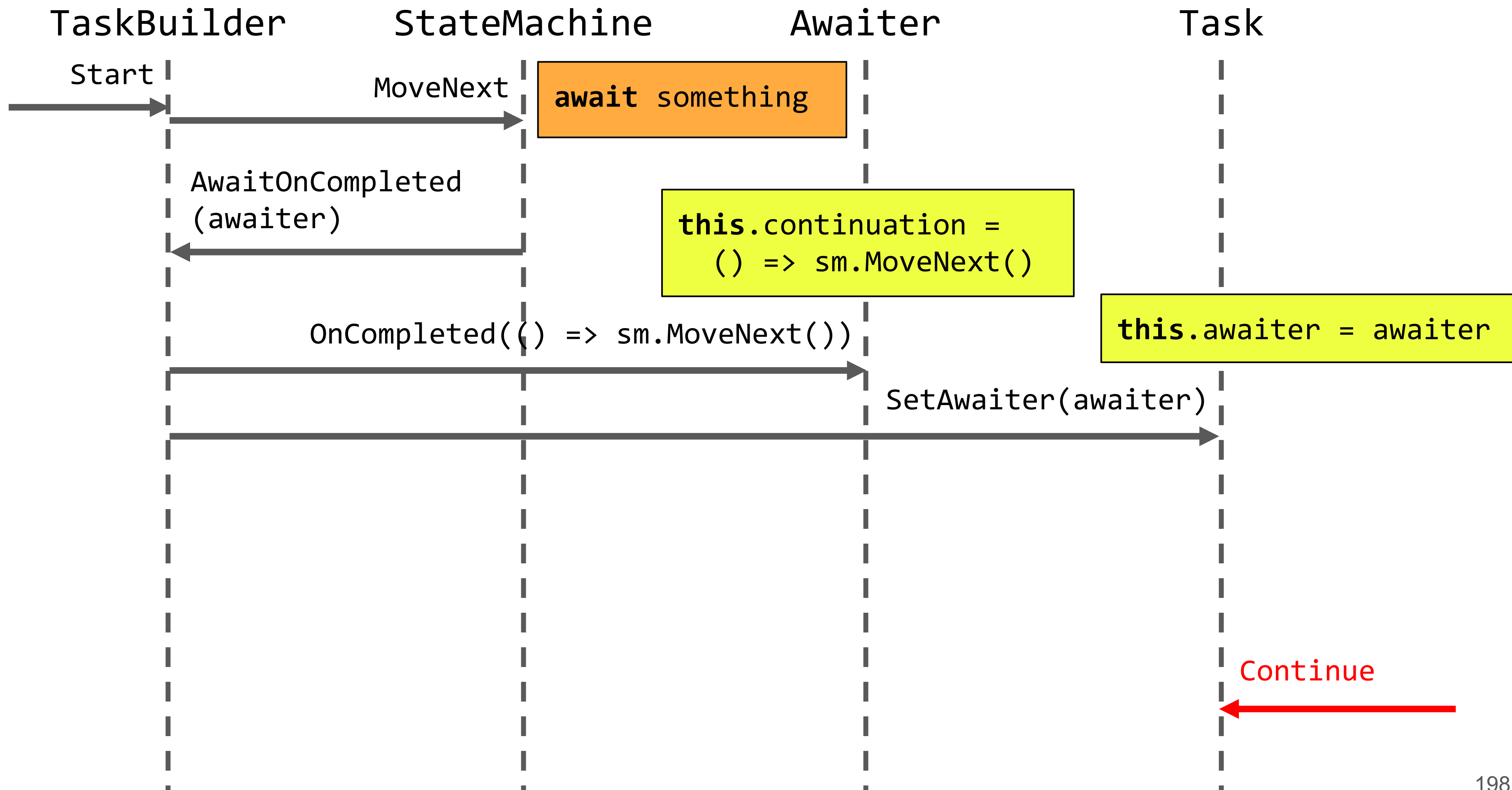

TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
OnCompleted(() => sm.MoveNext())

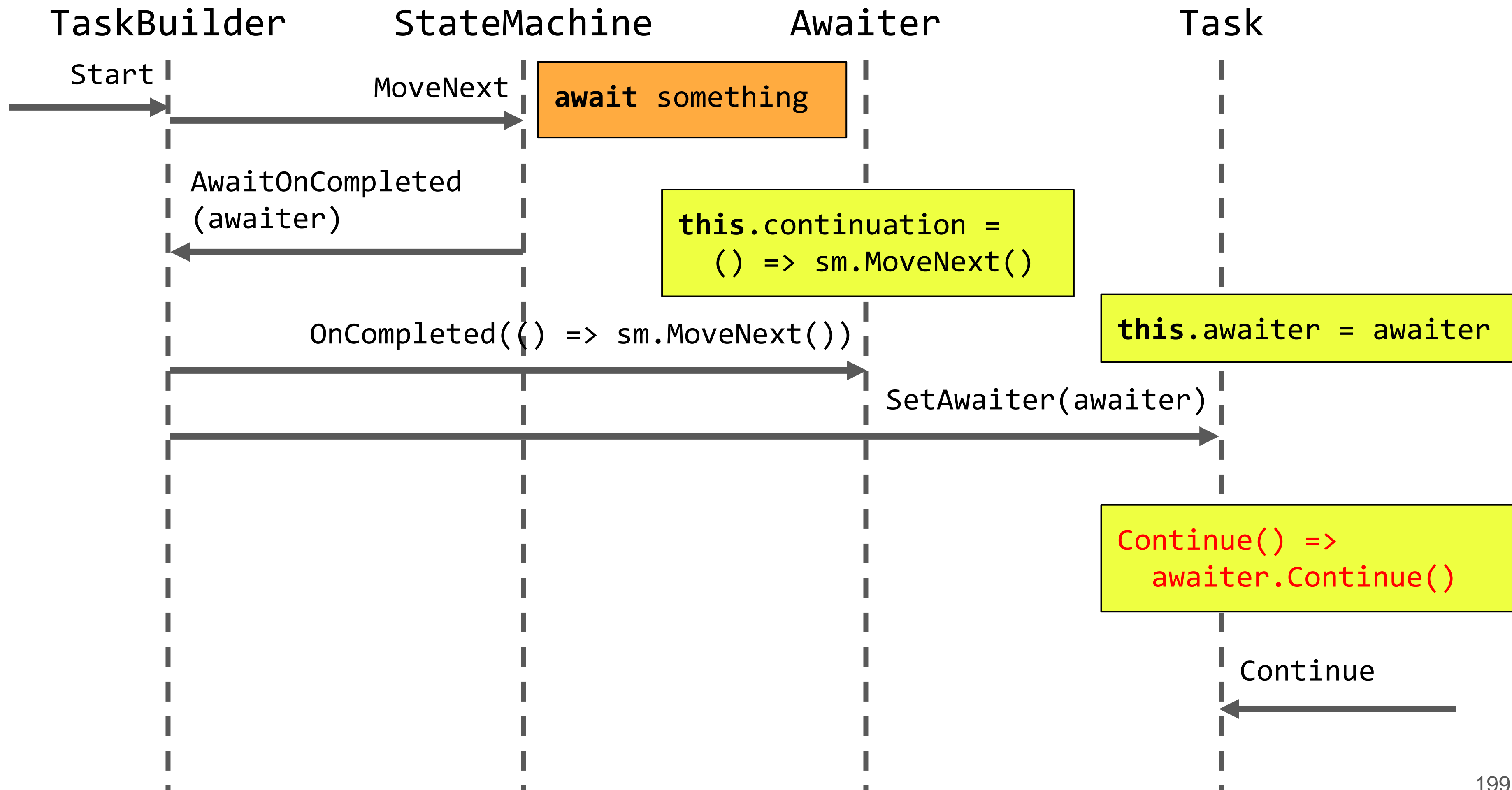

TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
OnCompleted(() => sm.MoveNext())

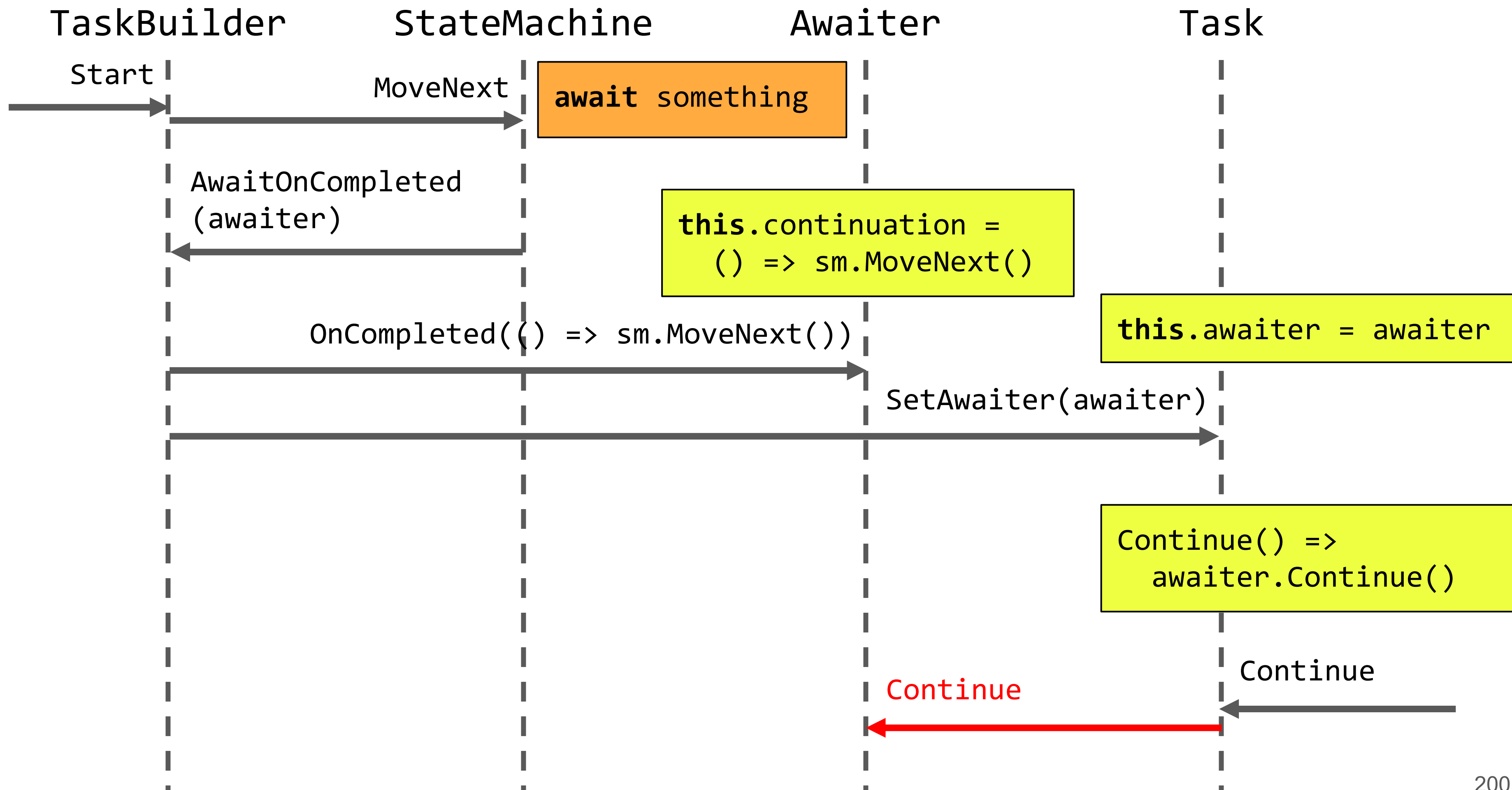

TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
OnCompleted(() => sm.MoveNext())
SetAwaiter(awaiter)



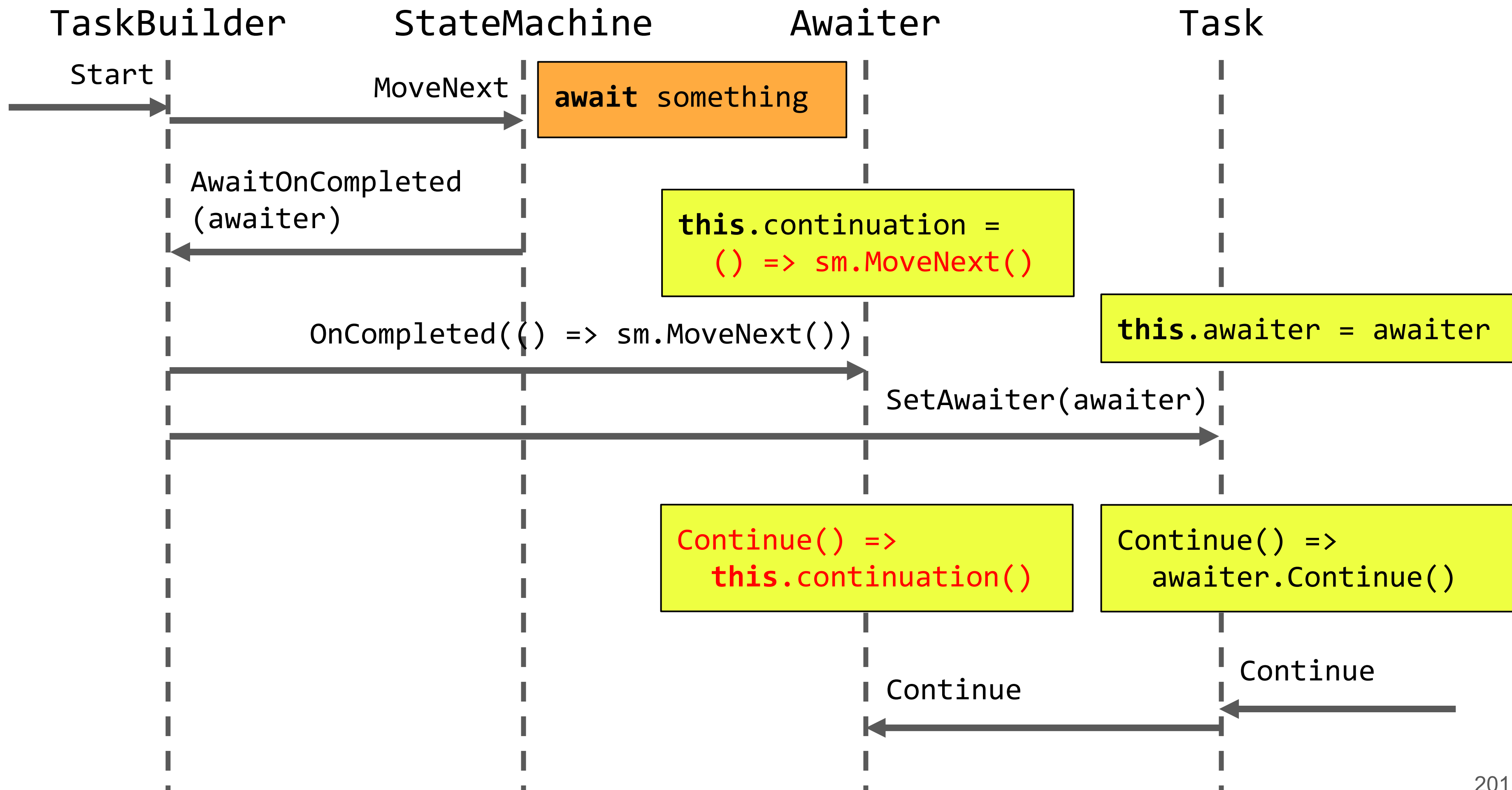
TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
this.awaiter = awaiter
OnCompleted(() => sm.MoveNext())
SetAwaiter(awaiter)

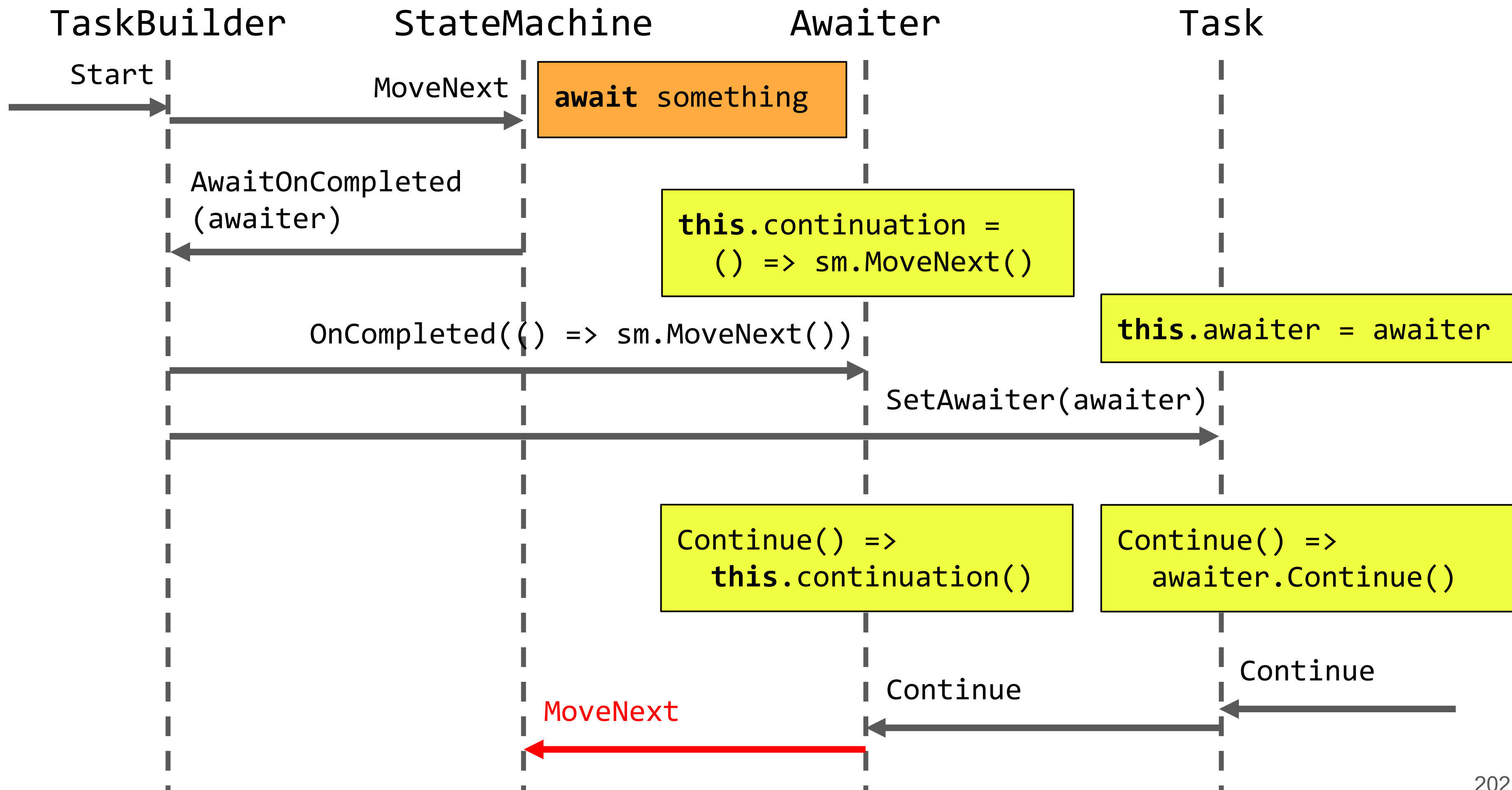

TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
this.awaiter = awaiter
OnCompleted(() => sm.MoveNext())
SetAwaiter(awaiter)
Continue


TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
this.awaiter = awaiter
OnCompleted(() => sm.MoveNext())
SetAwaiter(awaiter)
Continue() =>
awaiter.Continue()
Continue



TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
OnCompleted(() => sm.MoveNext())
this.awaiter = awaiter
SetAwaiter(awaiter)
Continue() =>
awaiter.Continue()
Continue
Continue


TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
OnCompleted(() => sm.MoveNext())
this.awaiter = awaiter
SetAwaiter(awaiter)
Continue() =>
this.continuation()
Continue() =>
awaiter.Continue()
Continue
Continue



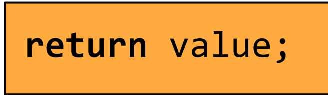
TaskBuilder
StateMachine
Awaiter
Task
Start
MoveNext
await something
AwaitOnCompleted
(awaiter)
this.continuation =
() => sm.MoveNext()
OnCompleted(() => sm.MoveNext())
this.awaiter = awaiter
SetAwaiter(awaiter)
Continue() =>
this.continuation()
Continue() =>
awaiter.Continue()
Continue
Continue
MoveNext

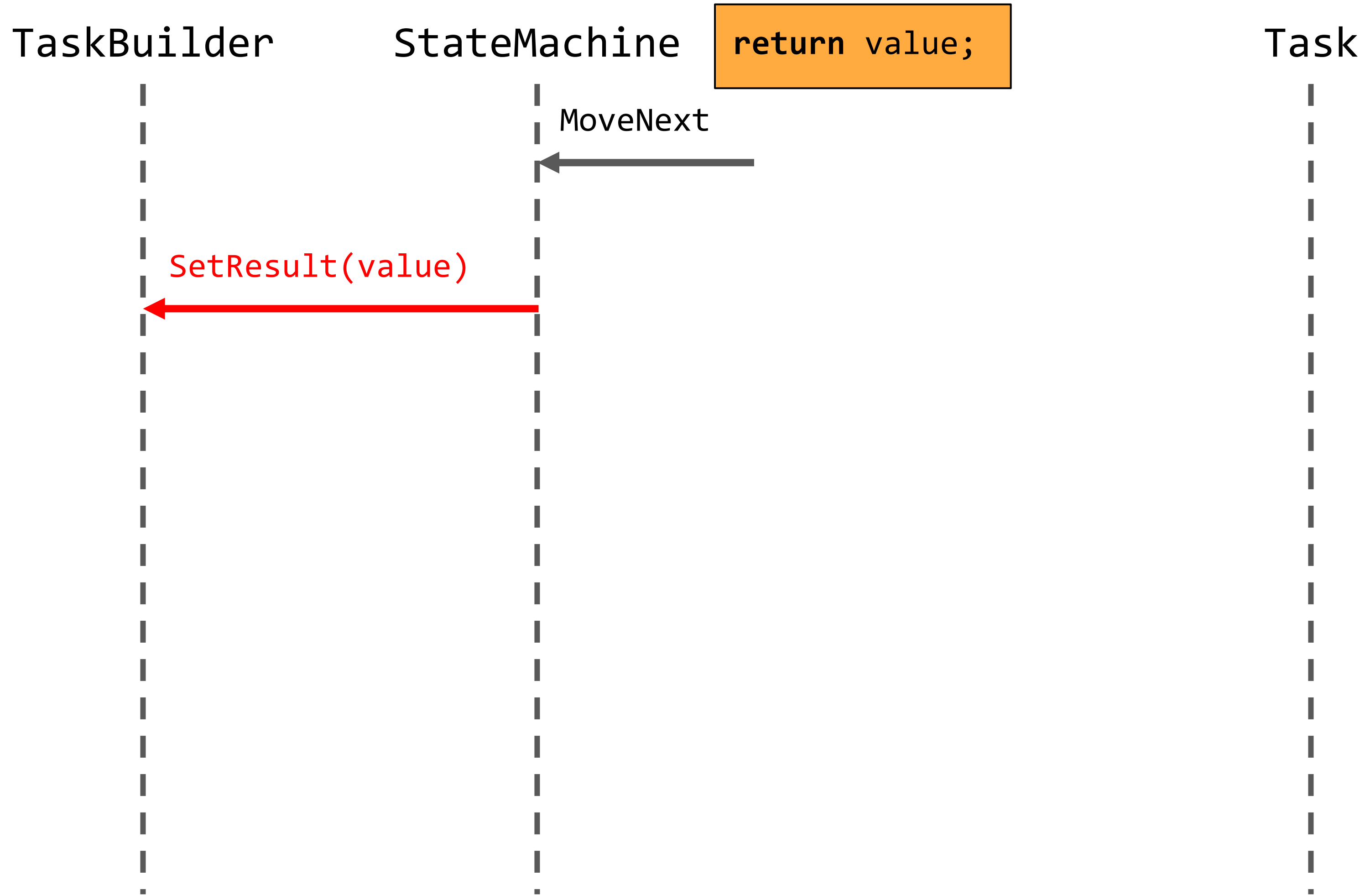
TaskBuilder StateMachine Task

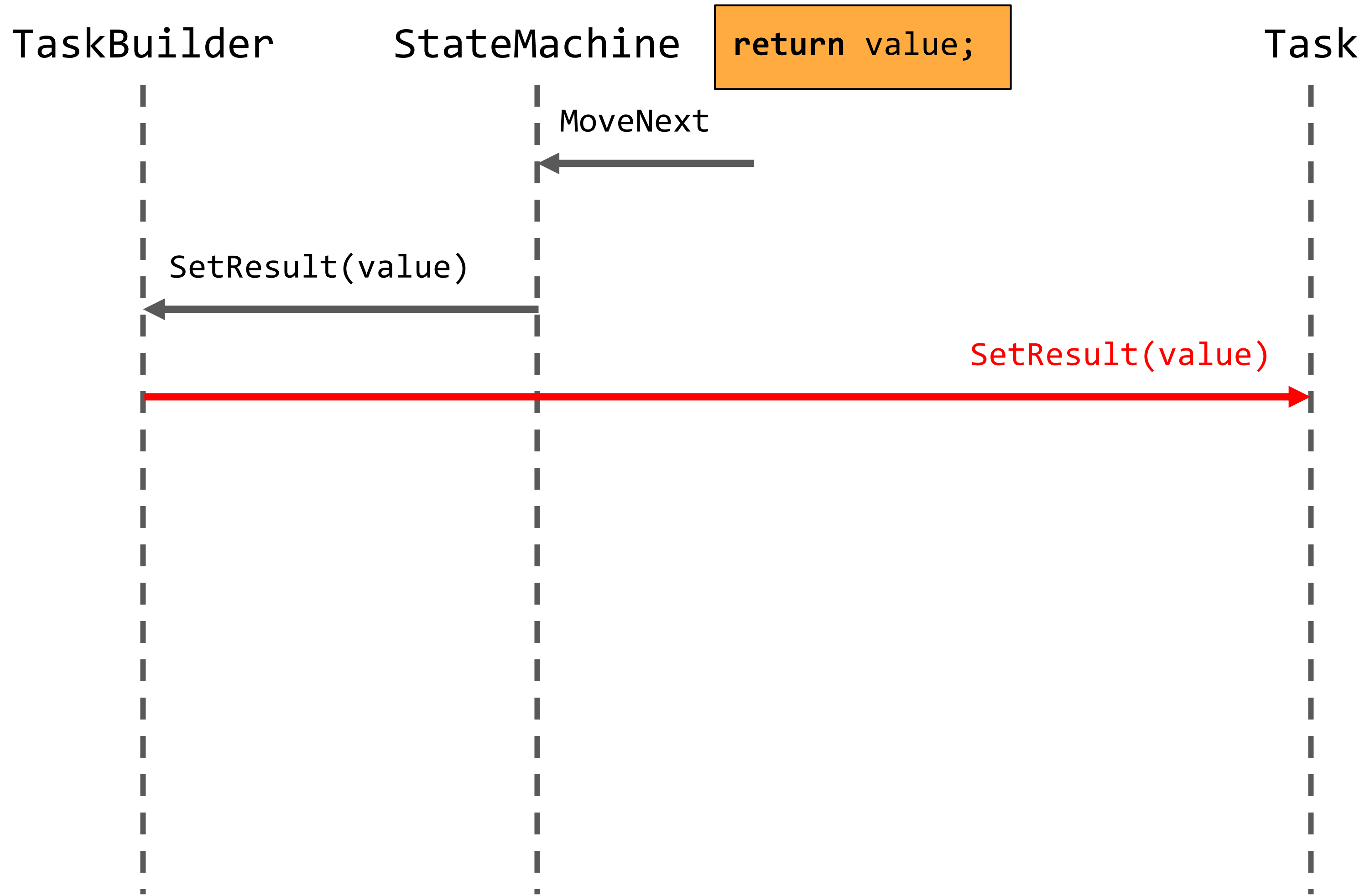
TaskBuilder StateMachine Task

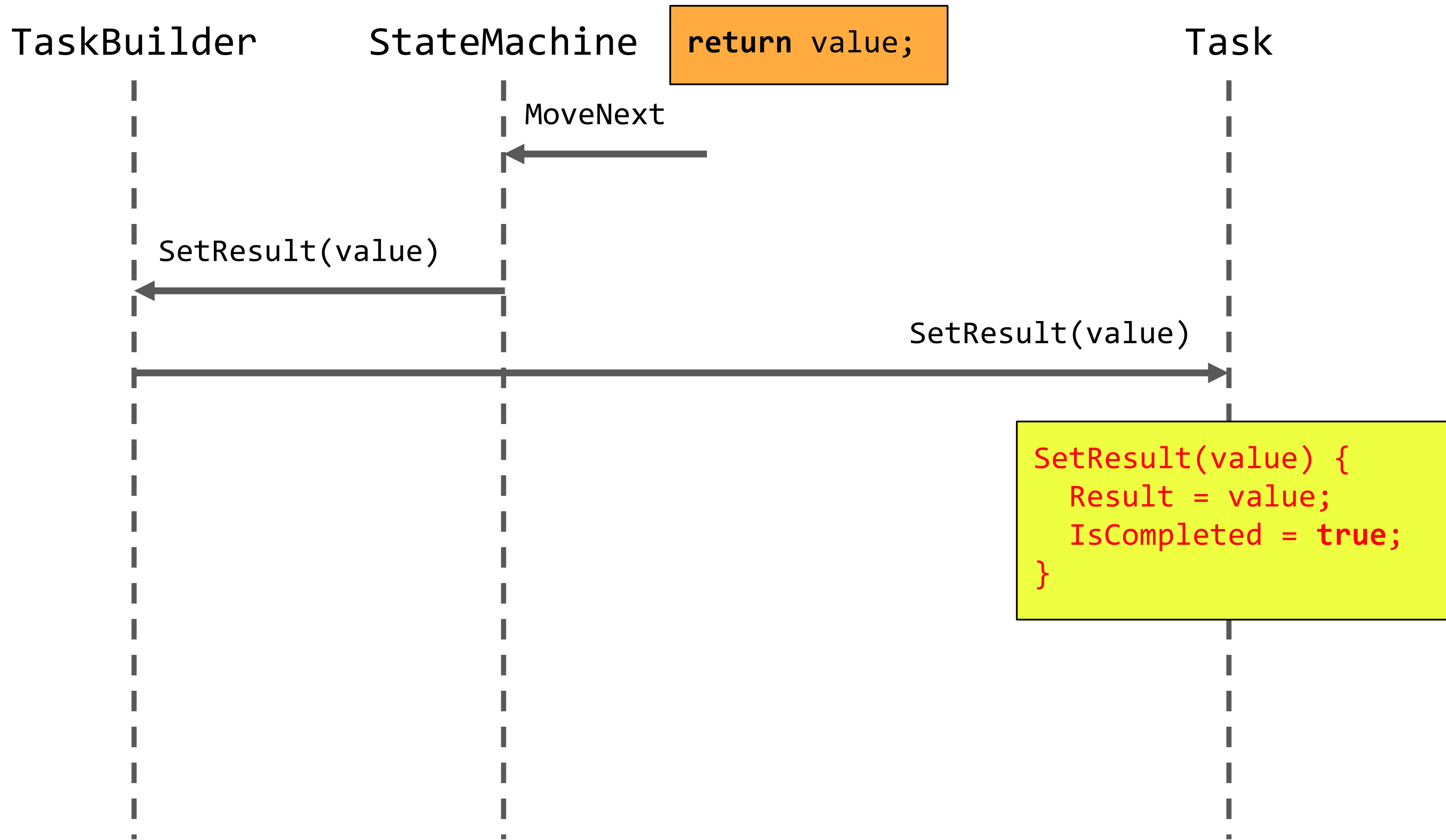
```
return value;
```

TaskBuilder StateMachine Task




TaskBuilder
StateMachine
return value;
Task
MoveNext
SetResult(value)


TaskBuilder
StateMachine
return value;
Task
MoveNext
SetResult(value)
SetResult(value)


TaskBuilder
StateMachine
return value;
Task
MoveNext
SetResult(value)
SetResult(value)
SetResult(value) {
Result = value;
IsCompleted = true;
}

Чего не хватает

Чего не хватает

- `WhenAny`, а не только `WhenAll`

Чего не хватает

- `WhenAny`, а не только `WhenAll`
- Вызов других асинхронных методов:

```
async StrategyTask<bool> Run() {
  int x = await HelperMethod();
}

async StrategyTask<int> HelperMethod() {
  ...
}
```

Что получилось

Строительство по генеральному плану

- Разбиваем фигуру на регионы

Строительство по генеральному плану

- Разбиваем фигуру на регионы
- Размножаем ботов

Строительство по генеральному плану

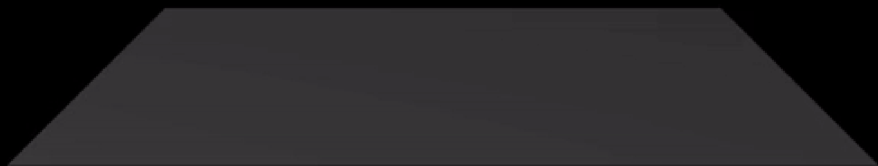
- Разбиваем фигуру на регионы
- Размножаем ботов
- Множество строящихся регионов - изначально пустое

Строительство по генеральному плану

- Разбиваем фигуру на регионы
- Размножаем ботов
- Множество строящихся регионов - изначально пустое
- Выбираем подходящий для строительства регион:
  - Не падает
  - Не мешает строить другие

Строительство по генеральному плану

- Разбиваем фигуру на регионы
- Размножаем ботов
- Множество строящихся регионов - изначально пустое
- Выбираем подходящий для строительства регион:
  - Не падает
  - Не мешает строить другие
- Все незанятые боты разбегаются кто куда



Как это применить?

Кооперативная многозадачность

```
class Enemy {
  async MicroTask Patrol() {
    while (alive) {
      if (CanSeeTarget()) {
        var attackSucceeded = await Attack();
        ...
      }
      else {
        MoveTowardsNextPoint();
        await TimeSpan.FromSeconds(1);
      }
    }
  }
}
```

Кооперативная многозадачность

- Есть реализации на `yield return`, но нет на `async/await`

https://mhut.ch/journal/2010/02/01/iterator-based-microthreading

Кооперативная многозадачность

- Есть реализации на `yield return`, но нет на `async/await`

  https://mhut.ch/journal/2010/02/01/iterator-based-microthreading

- Фронтендеры уже делают это - `redux-saga`

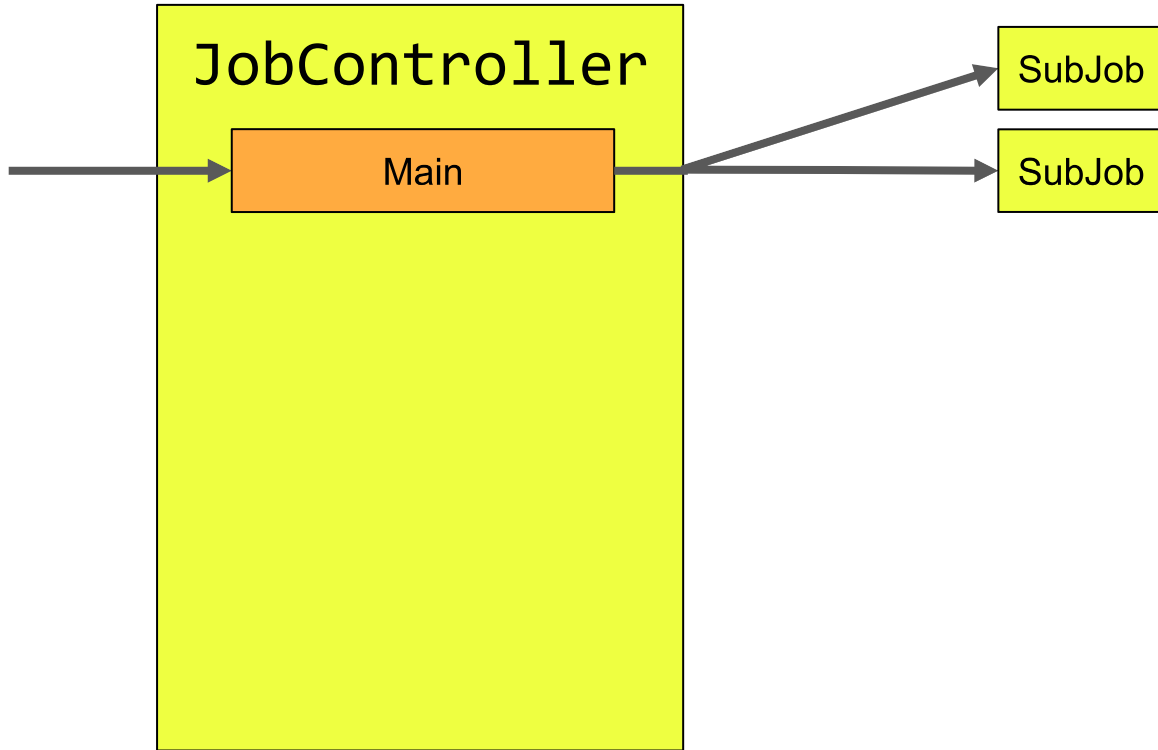
```
a = yield b; // у них так можно!
```

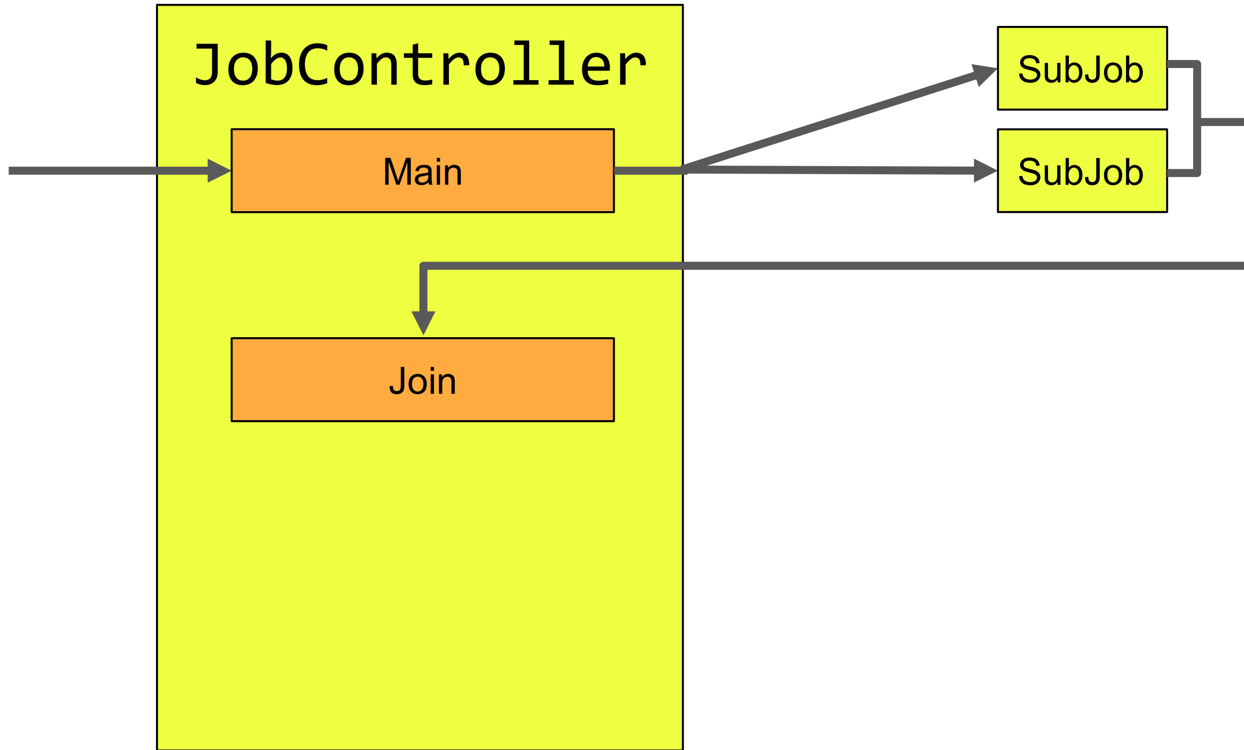
Distributed fork-join state machine

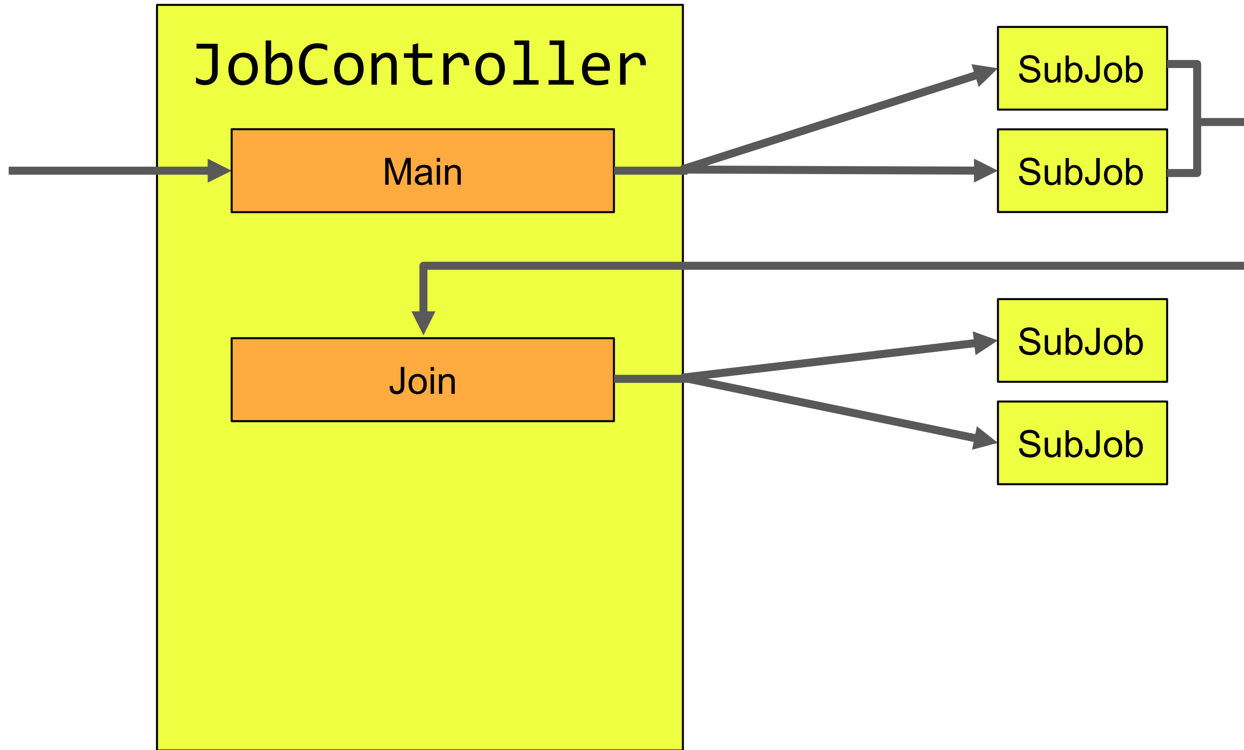
JobController

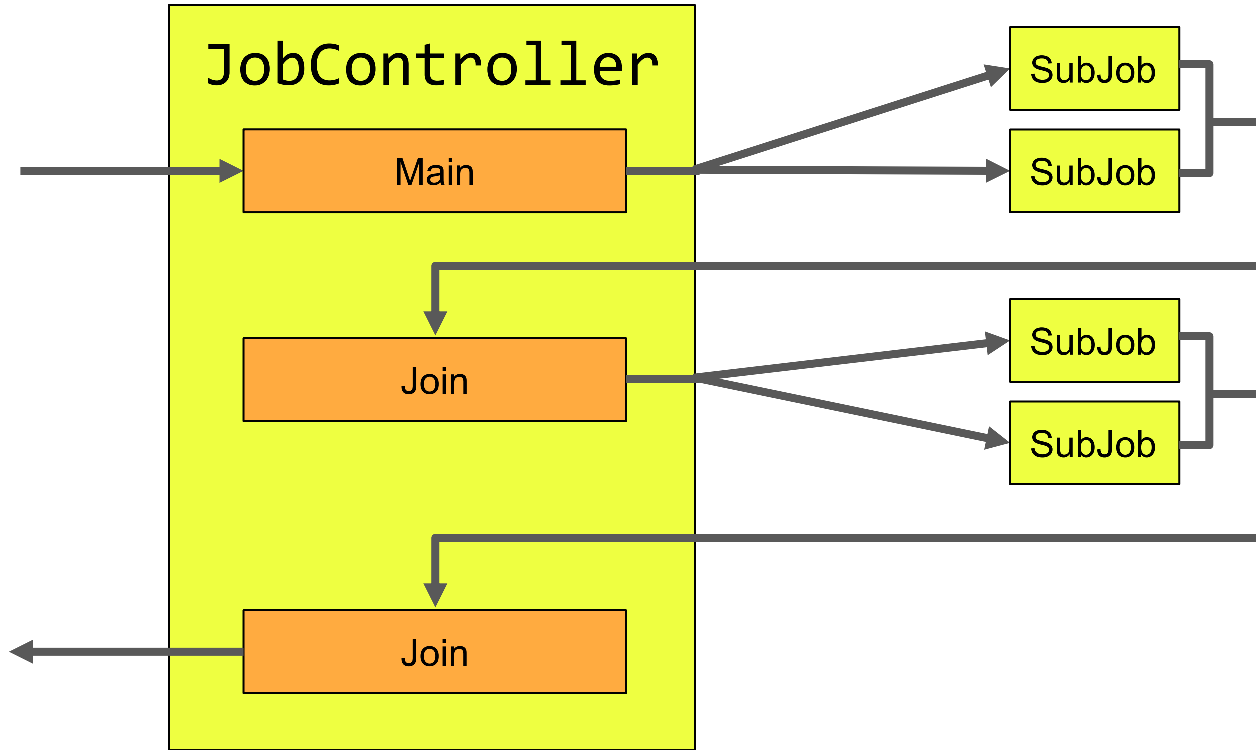
JobController

Main


JobController
Main
SubJob
SubJob


JobController
Main
Join
SubJob
SubJob


JobController
Main
Join
SubJob
SubJob
SubJob
SubJob


JobController
Main
Join
Join
SubJob
SubJob
SubJob
SubJob

```
class MyJobController {

  async JobTask<int> Handle() {

    var b = await SubJob("x", 10);

    if (b > 10)
      return await SubJob("y", b)

    return 10 + await SubJob("z", b)
  }
}
```

Исследуйте технологии, которые используете, порой им можно найти самые неожиданные применения!

Полезные ссылки

- Репозиторий нашей команды на гитхабе - https://github.com/kontur-contests/icfpc2018-kontur-ru/tree/dotnext
- Цикл статей про асинхронные методы - https://blogs.msdn.microsoft.com/seteplia/2017/11/30/dissecting-the-async-methods-in-c/
- Кооперативная многозадачность на итераторах - https://mhut.ch/journal/2010/02/01/iterator-based-microthreading

Вопросы?