１、序論

本レポートの目的は Faatory によって作成されたオブジェクトであっても State パターンの恩恵を得ることができることを示すことである。

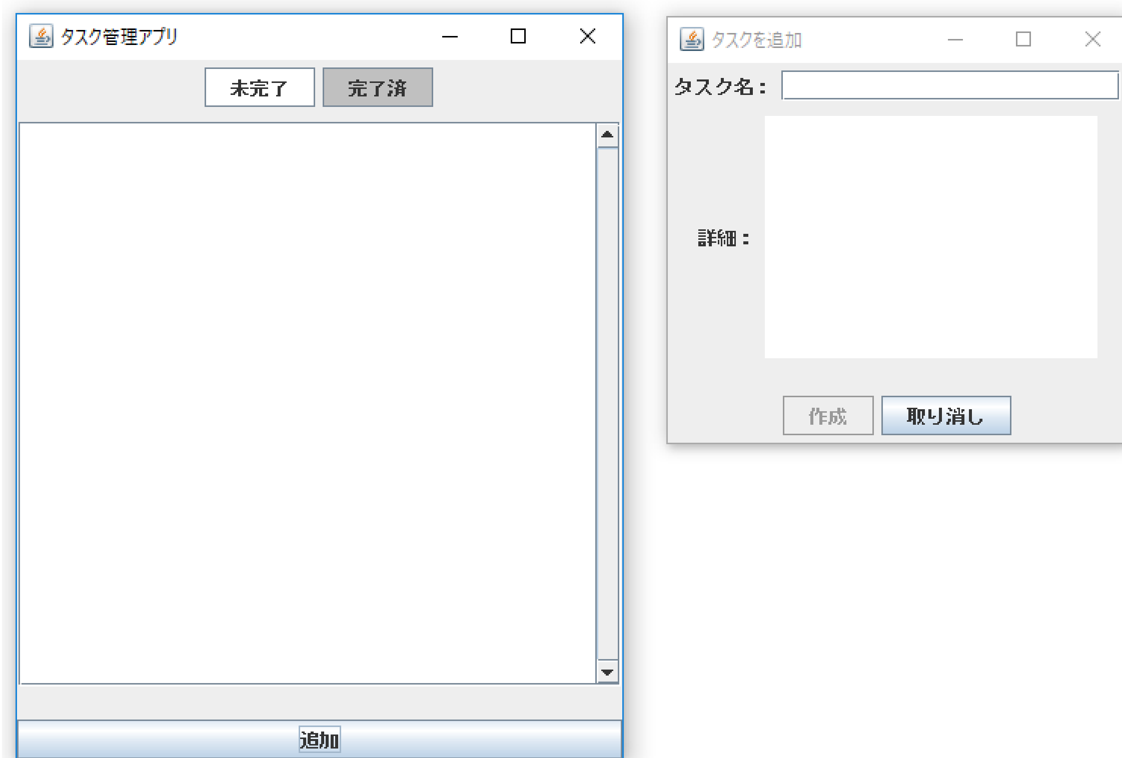
そのため、Factory Method パターンと State パターンを組み合わせて単純なタスク管理アプリを作成する。State パターンによって達成、未達成の状態を表現できる。タスクはユーザーの操作で増えるので、タスクのオブジェクトを作成する工場を実装する。これにより、オブジェクト A は状態 α だがオブジェクト B は状態 β のように、複数のオブジェクトがそれぞれ状態を持つ場合にも State パターンが利用できることを確認する。

Factory パターンは、インスタンス作成のための枠組みと、実際のインスタンス作成のクラスを分けて考えることができるようになる。State パターンは状態をクラスとして表現することで、状態の変化をクラスの切り替えで表現でき、新しい状態を追加しなければならないときに何を実装する必要があるのかを把握できる。

２、サンプルプログラム

２－１、サンプルプログラムの概要

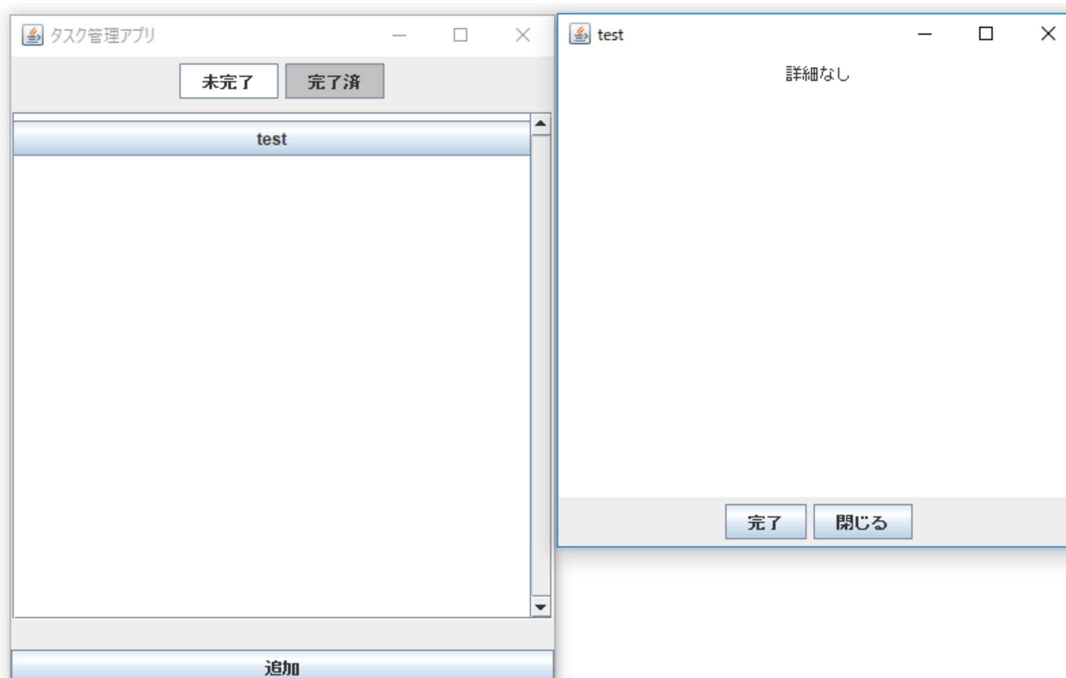
本プログラムでは、ユーザーの操作を受け取り「タスク」の作成と完了と詳細の確認ができるソフトウエアを作成する。「タスク」とはタスク名と詳細の二つの文字列を保持し、完了済あるいは未完了のどちらかが識別できるオブジェクトであるとする。



メイン画面（左）とタスク追加画面（右）

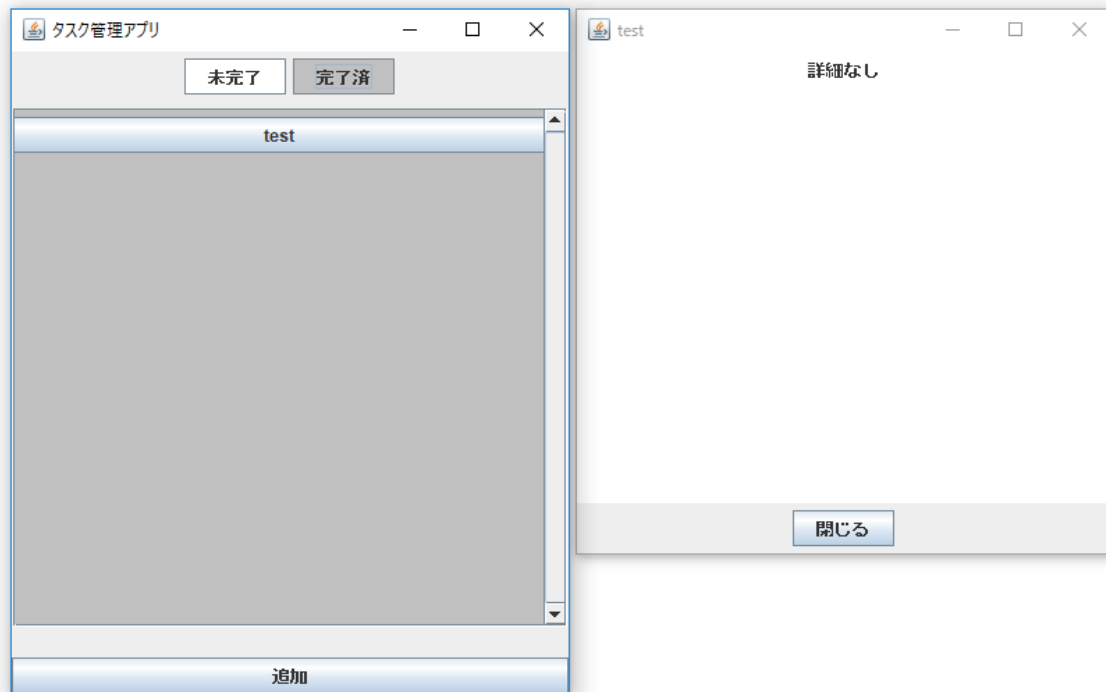
ユーザーはメイン画面から追加ボタンを押すことでタスク追加画面を表示する。タスク追加画面では、タスク名を入力して作成ボタンを押すことでメイン画面の未完了パネルにタスクを追加することができる。このとき、タスク名は必須だが詳細は未入力でも構わないとする。

実際にタスク名は test、詳細は入力せずに作成したサンプルが以下である。



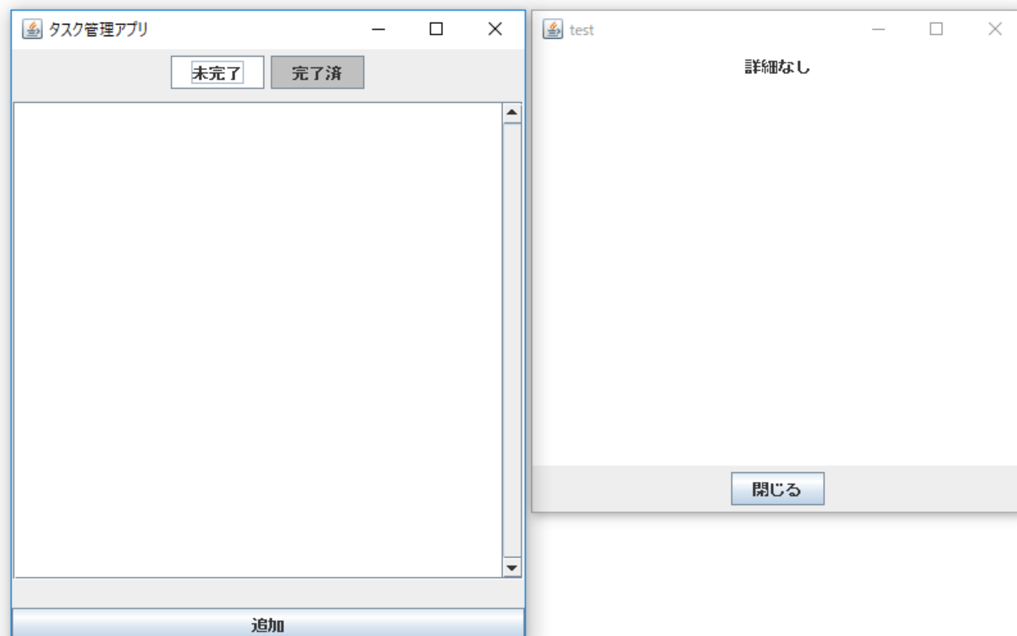
タスク名 test で追加したメイン画面（左）とタスク test の詳細画面（右）

作成ボタンを押すと、追加画面は閉じ、メイン画面では未完了のタスクの一覧が表示されている。続いて、このタスクを完了する。単純に、詳細画面の下の完了ボタンを押すのみである。



完了直後のメイン画面（左）と test の詳細画面（右）

完了ボタンを押すと、詳細画面は閉じ、メイン画面は完了済みのタスクの一覧が表示される。完了済みのタスクの詳細を開くと、上記の通り完了ボタンが存在しない詳細画面が表示される。

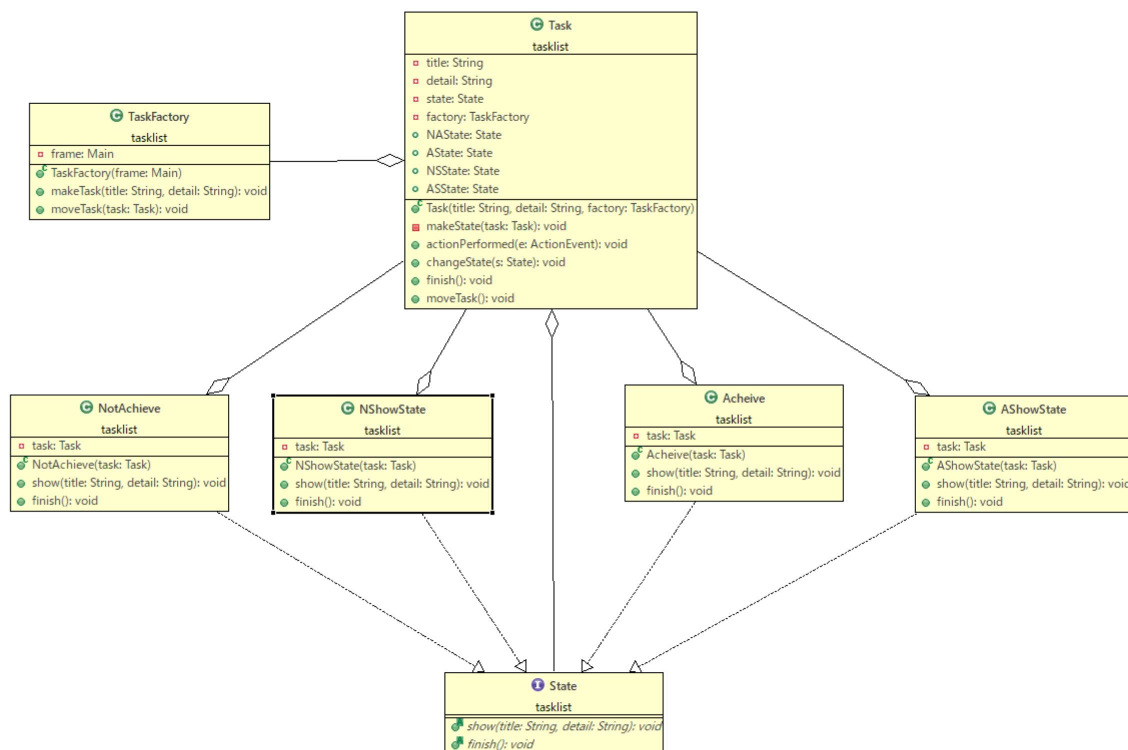


未完了の一覧を表示しているメイン画面（左）と test の詳細画面（右）

自動で遷移する以外にも、メイン画面上部のボタンによって、未完了の一覧と完了済みの一覧のどちらを表示するかを選択することができる。

２－２、Task の概要

Task の State は NotAchieve（未完了かつ詳細画面を表示していない）、NShowState（未完了かつ詳細画面を表示中）、Achieve（完了済かつ詳細画面を表示していない）、AshowState（完了済かつ詳細画面を表示している）の 4 種類である。クラス図の作成には AmaterasUML[1]という Eclipse プラグインを入手して利用した。



サンプルプログラムのタスクに関する一部のクラス図

本プログラムでは、タスクは GUI のボタンとして実装したので JButton を継承している。タスクは完了済、未完了の状態に加えて詳細を開いているか否かの状態も管理するために両者の直積の 4 つの状態を取る。状態の Show メソッドは詳細画面を作成し、finish メソッドは完了することを目的とするが、状態によっては「何もしない」メソッドを実装している。これによって、一つのタスクが複数の詳細画面を作成することを防止している。

これらの状態を持つタスクは工場によって作られる。工場は自身の親会社にあたるメインフレームをフィールドとして保持している。タスクの作成とメインフレームへの登録を行う makeTask メソッドと、タスクの部署替えを打診する moveTask メソッドを持つ。これらの仕組みにより、Task 自身はメインフレームのことを知らないが、自分を作った工場に申請することでメインフレームに影響を与えることができる。

State.java

```
package tasklist;
```

```
public interface State {
    public abstract void show(String title,String detail);
    public abstract void finish();
}
```

State インターフェースは画面を表示する show メソッドと状態を終了する finish メソッドの実装を約束しているのみである。

NotAchieve.java

```
package tasklist;

public class NotAchieve implements State {
    public NotAchieve(Task task) {
        this.task=task;
    }
    private Task task;



    @Override
    public void show(String title,String detail) {
        // TODO 自動生成されたメソッド・スタブ
        new NotAcheiveDetailFrame(title,detail,this.task);
        this.task.changeState(this.task.NSState);

    }

    @Override
    public void finish() {
        // TODO 自動生成されたメソッド・スタブ

    }

}
```

未完了の状態を表す NotAchieve クラスでは State インターフェースを実装している。このコンストラクタでは状態に紐づける Task のインスタンスを保持する。この状態から直接完了済み状態に移行することはないので、finish メソッドは何もしない。詳細画面を開かずに状態を完了済みに移行する機能を追加するときにはここに加筆した上でこのメソッドを呼

び出すようにすればよい。show メソッドは詳細画面を作成し、未完了かつ詳細画面を開いている NShowState に状態を切り替える。

NShowState.java

```
package tasklist;
public class NShowState implements State {
    private Task task;
    public NShowState(Task task) {
        this.task=task;
    }
    @Override
    public void show(String title, String detail) {
        // TODO 自動生成されたメソッド・スタブ
    }
    @Override
    public void finish() {
        // TODO 自動生成されたメソッド・スタブ
        this.task.changeState(this.task.AState);
        this.task.moveTask();
    }
}
```

完了かつ詳細画面を開いている状態である NShowState では、それ以上に詳細画面を開くことはないので show メソッドでは何もしない。逆に、詳細画面の完了ボタンを押すことで完了済状態に移行するため finish メソッドに処理が定義されている。完了ボタンを押さずに詳細画面を閉じたときは NotAchieve 状態に戻る。この時閉じるボタンを押した場合のみでなくウィンドウのバツボタンを押して消した場合にも問題なく遷移する[2]。

Task.java

```
package tasklist;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Task extends TaskButton implements ActionListener {
    private String title;
    private String detail;
    private State state;
```

```
private TaskFactory factory;

public State NAState;
public State AState;
public State NSState;
public State ASState;


public Task(String title,String detail,TaskFactory factory) {
    super(title);
    this.title=title;
    this.detail=detail;
    this.addActionListener(this);
    this.factory=factory;
    makeState(this);
    // TODO 自動生成されたコンストラクター・スタブ
}
private void makeState(Task task) {
    NAState=new NotAchieve(task);
    AState=new Acheive(task);
    NSState=new NShowState(task);
    ASState=new AShowState(task);
    state=NAState;
}

@Override
public void actionPerformed(ActionEvent e) {
    // TODO 自動生成されたメソッド・スタブ
    if(e.getSource()==this) {
        state.show(title, detail);
    }
}
public void changeState(State s) {
    this.state=s;
}
public void finish() {
```

```
        this.state.finish();
    }
    public void moveTask() {
        this.factory.moveTask(this);
    }

}
```
このサンプルの要である Task クラスである。Task はボタンを継承しているので GUI のパーツとして扱うことができる。Task のインスタンスはタスク名を保持する title、詳細を保持する detail、現在の状態を保持する state、作成元の工場を保持する factory をプライベートなフィールドとして持つ。そのほかに自身と紐づいた各種状態のインスタンスを保持するパブリックなフィールドを持つ。

Task のコンストラクタでは、引数を適当なフィールドに設定することとアクションリスナーを設定すること、各種状態のインスタンスを作成し初期状態である NotAchive 状態を設定しておくことを行っている。

Task はボタンである。自身が押されたときは自身の今の状態が何であるかを意識することなく show メソッドを呼び出す。また、moveTask メソッドでは自身を作成した factory のメソッドを呼び出す。これにより Task ボタンは自身の組み込まれた GUI の詳細を把握することなく GUI からの取り外し、取り付けなどを factory に要請することができる。

TaskFactory.java
```
package tasklist;
public class TaskFactory {
    private Main frame;
    public TaskFactory(Main frame) {
        this.frame=frame;
    }
    public void makeTask(String title,String detail) {
        Task a=new Task(title,detail,this);
        this.frame.Paneladd(a);
    }
    public void moveTask(Task task) {
        this.frame.Panemove(task);
    }
}
```
TaskFactory は単純な Task の作成だけでなく、作成した Task とフレームとの橋渡しも担当する。そのため、Factory は自分の影響するフレームをフィールドとして保持する。

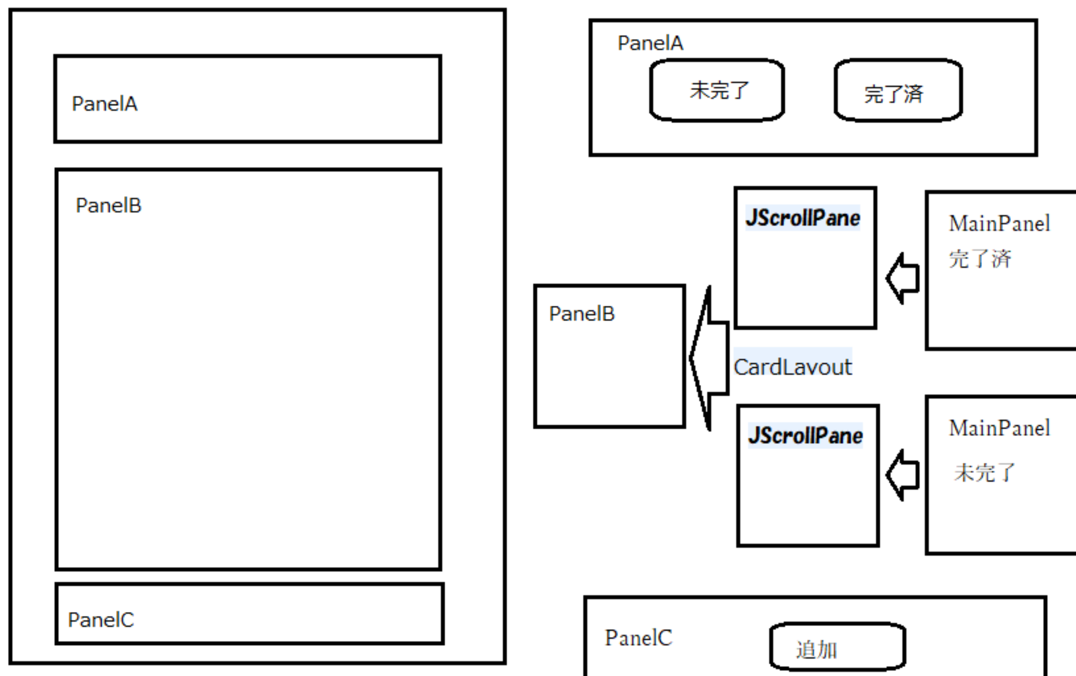
２－３、TaskButton

Task は JButton を継承した TaskButton を継承している、TaskButton は単純に見た目を整えているだけである。よって Task はボタンであり、ボタンを押されたときの挙動を定義している。それは、現在の状態の Show メソッドを呼び出すという単純なものである。

Show メソッドは NShowState と AshowState ではなにもしない。NotAchieve では完了ボタン付きの詳細画面を表示して状態を NShowState に移行する。Achieve では完了ボタンのない詳細画面を表示して状態を AshowState に移行する。

Task はボタンであるつまり Java の GUI のパーツとして扱うことができる。そのためにメイン画面において、パネルに追加したり除いたりを自在に行うことができる。

２－４、サンプルプログラムの GUI



メイン画面の GUI の概要図

メイン画面は一つのフレームの上に３つのパネルが乗っていて、それぞれ BorderLayout によって上中下に配置されている。上下のパネルは単純にボタンが設置されているだけである。一方で中央のパネルは少し複雑で、CardLayout にて表現されている。中央のパネルには完了済みのタスクのボタンを保持する「完了済みメインパネル」と未完了のタスクボタンを保持する「未完了メインパネル」の２種類が、それぞれスクロールバーを実装するパネルで修飾されてカードとして登録されている。２種類のカードはプログラムの動作の他、上部のボタンでも切り替えることができる。

タスク追加画面は mediator パターンを利用し、タスク名が未入力の場合は作成ボタンを押せないようにしてあるが、今回の趣旨とは外れるので深くは触れないでおく。GUI の設計、実装に関して多くの文献を参考にした[3][4]。

３、Factory と State パターン

Factory と State パターンがそれぞれ有効に働いていることを確認する。

タスクは状態をクラスで持つオブジェクトである。そのために Task は自身の状態を問わず show や finish メソッドを実行できる。もちろんその結果は状態によって変化する。if によって状態を識別せずに挙動を制御できるのは State パターンの恩恵である。「この State のインスタンスはこのタスクのものである」ということが示されるように、タスクは現在の状態とは別に 4 つの State を保持するフィールドを持つ。そうすることで、Factory によって次々とタスクが作成され、中身はほとんど同じである State クラスが作成されてもそれぞれは独立して扱うことができる。

タスクは Factory によって作成されるが、この Factory がメイン画面と別れていることで、タスクはメイン画面のことを知ることなくふるまうことができる。

４、結論

Factory によって作成されたオブジェクトであっても、State の依存関係を明示すれば State パターンの恩恵を得ることができた。

5、参考サイト

[1] AmaterasUML
amateras.osdn.jp/cgi-bin/fswiki/wiki.cgi?page=AmaterasUML
[2] java で右上の閉じるボタンではなく、JButton でボタンを実装し、そのボタンを押すとウインドウを閉じるプログラムを作りたいのですが、その場合イベントリスナーの部分は...
https://detail.chiebukuro.yahoo.co.jp/qa/question_detail/q1175482206
[3] Let's プログラミング
https://www.javadrive.jp/tutorial/
[4] よく使いそうなレイアウト：BoxLayout
https://shoken.hatenablog.com/entry/20070717/1184676277