```
package player;

import player.PlayerI.*;
import player.playeragent.*;
import edu.neu.ccs.demeterf.demfgen.lib.List;
import gen.*;

/**
 * Takes the player's turn.
 */
public class PlayerRunner {
    private Player player;
    private PlayerI genericPlayer;

    public PlayerRunner(String pid, PlayerI play) {
        genericPlayer = play;
        player = new Player(new PlayerID(Integer.parseInt(pid)), play.getName());
    }

    /**
     * Forwarded from static to dynamic.
     */
    public void main() {
        Printer p = new Printer(Printer.Caller.MAIN);
        p.start();

        List<Transaction> trans = buyOrReofferDerivative().append(
                createDerivative().append(
                        deliverRawMaterial().append(finishProduct())));

        Util.commitTransaction(new PlayerTransaction(player, trans));

        p.print("\n<strong>Account:</strong> "
                + String.valueOf(Util.getAccount(player)));

        p.end();
    }

    /**
     * This one shouldn't need to be changed.
     */
    public ReofferAgent getReofferAgent() {
        return new ReofferAgent();
    }

    /**
     * Wrap a derivative into a transaction of the given transaction type.
     */
    private class TransWrap extends List.Map<Derivative, Transaction> {
        private TransactionType type;

        public TransWrap(TransactionType t) {
            type = t;
        }

        public Transaction map(Derivative d) {
            return new Transaction(type, d);
        }
    }

    /**
     * Creates a derivative.
     */
    private List<Transaction> createDerivative() {
        List<Transaction> result = List.<Transaction> create();

        Printer p = new Printer(Printer.Caller.CREATE);
        p.start();

        Derivative der = genericPlayer.getCreateAgent().createDerivative(
                player, Util.existingTypes());

        p.print(der.print());

        result = result.append(new Transaction(new Create(), der));

        p.end();

        return result;
    }
```

```
    /**
     * Buys a derivative from the sale stores or reoffers all of them.
     */
    private List<Transaction> buyOrReofferDerivative() {
        List<Transaction> result;

        Printer p = new Printer(Printer.Caller.BUY);
        p.start();

        List<Derivative> forSale = Util.forSale(player.id);

        if (forSale.isEmpty()) {
            p.print("Nothing to buy");
            p.end();

            return List.create();
        }

        double account = Util.getAccount(player);

        List<Derivative> bought = genericPlayer.getBuyAgent().buyDerivatives(
                forSale, account);

        if (!bought.isEmpty()) {
            for (Derivative d : bought) {
                p.print(d.print());
            }

            result = bought.map(new TransWrap(new Buy()));

            p.end();

            return result;
        }

        p.print("Reoffering all");

        result = reofferAll(forSale, player.id);

        p.end();

        return result;
    }

    /**
     * Returns a list of reoffered derivatives.
     */
    private List<Transaction> reofferAll(List<Derivative> forSale, PlayerID pid) {
        return Util.uniquelyTyped(forSale).map(
                new PriceReducer(getReofferAgent(), pid));
    }

    /**
     * Reduces an individual derivative using the reoffer agent.
     */
    private class PriceReducer extends List.Map<Derivative, Transaction> {
        ReofferAgent agent;
        PlayerID pid;

        PriceReducer(ReofferAgent a, PlayerID p) {
            pid = p;
            agent = a;
        }

        public Transaction map(Derivative d) {
            return new Transaction(new Reoffer(), agent.reofferDerivative(d,
                    pid));
        }
    }

    /**
     * Delivers raw material for the derivatives that need raw material.
     */
    private List<Transaction> deliverRawMaterial() {
        List<Transaction> result;

        Printer p = new Printer(Printer.Caller.DELIVER);
        p.start();

        List<Derivative> needRM = Util.needRawMaterial(player);
```

```
        for (Derivative d : needRM) {
            p.print(d.print());
        }

        result = needRM.map(new Deliverer(genericPlayer.getDeliverAgent()));

        p.end();

        return result;
    }

    /**
     * Handles the calling of deliver agent.
     */
    private class Deliverer extends List.Map<Derivative, Transaction> {
        DeliverAgentI agent;

        Deliverer(DeliverAgentI a) {
            agent = a;
        }

        /**
         * Call the deliver agent and wrap the transaction.
         */
        public Transaction map(Derivative d) {
            return new Transaction(new Deliver(), agent.deliverRawMaterial(d));
        }
    }

    /**
     * Finishes the derivatives that need finishing.
     */
    private List<Transaction> finishProduct() {
        List<Transaction> result;

        Printer p = new Printer(Printer.Caller.FINISH);
        p.start();

        List<Derivative> toFinish = Util.toBeFinished(player);

        result = toFinish.map(new Finisher(genericPlayer.getFinishAgent()));

        p.end();

        return result;
    }

    /**
     * Handles the calling of finish agent.
     */
    private class Finisher extends List.Map<Derivative, Transaction> {
        FinishAgentI agent;

        Finisher(FinishAgentI a) {
            agent = a;
        }

        /**
         * Call the finish agent and wrap the transaction.
         */
        public Transaction map(Derivative d) {
            return new Transaction(new Finish(), d.finish(agent
                    .finishDerivative(d)));
        }
    }
}
```