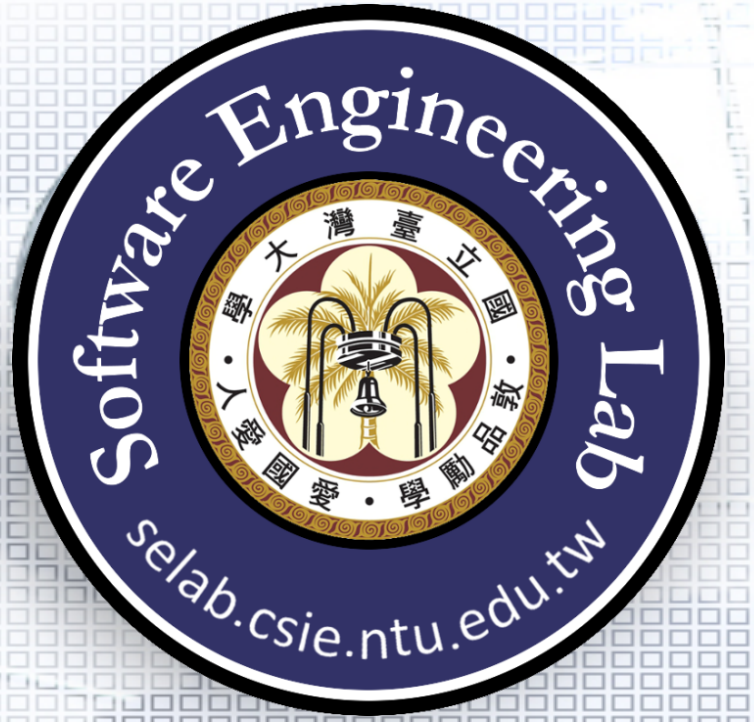
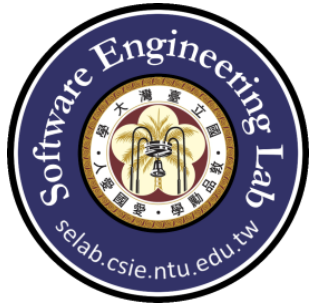


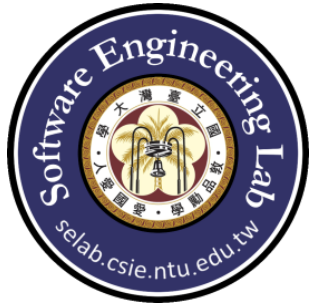
Singleton Pattern

Prof. Jonathan Lee（李允中）
Department of CSIE
National Taiwan University



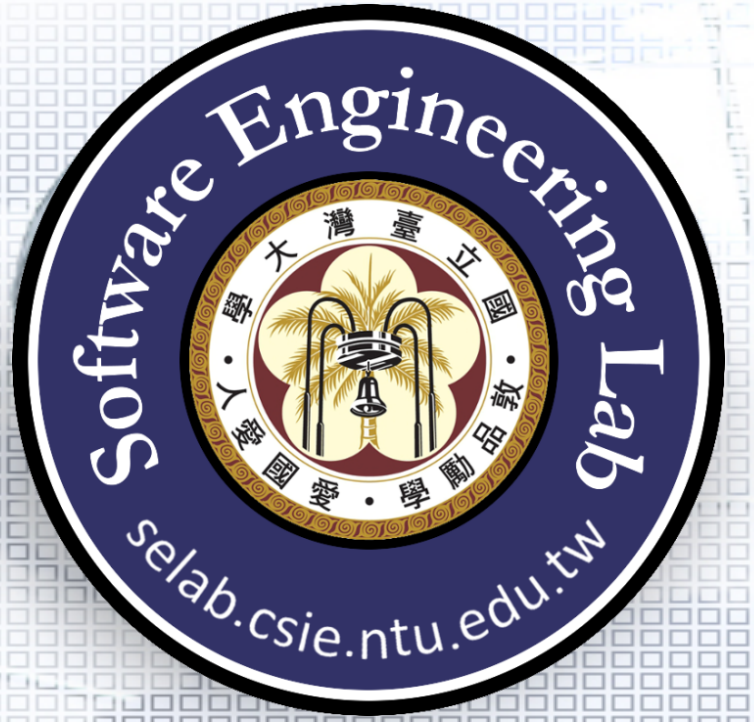
Design Aspect of Facade

the sole instance of a class



Outline

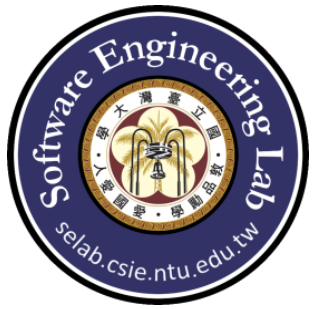
- Chocolate Boiler Requirements Statements
- Initial Design
- Recurrent Problems
- Intent
- Singleton Pattern Structure
- Singleton with Multi-threading Issues



Chocolate Boiler

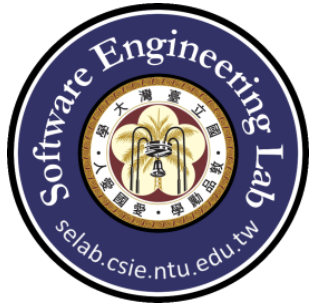
Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University



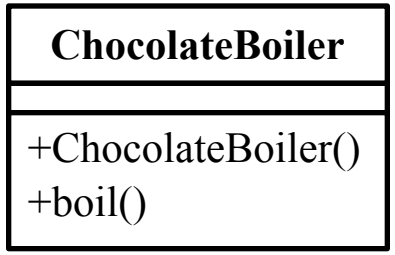
Requirements Statements

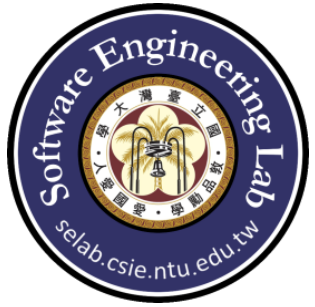
- A chocolate boiler is used to boil chocolate.
- Before boiling chocolate with the boiler, you have to make sure that the boiler is now empty and then fill chocolate in. Besides, you can't boil chocolate again while the chocolate has already been boiled.
- After boiling, drain out the boiled chocolate and make the boiler empty again.
- In order to prevent some unexpected situation, it is not allowed to have multiple instances of the chocolate boiler in the system.



Requirements Statements$_1$

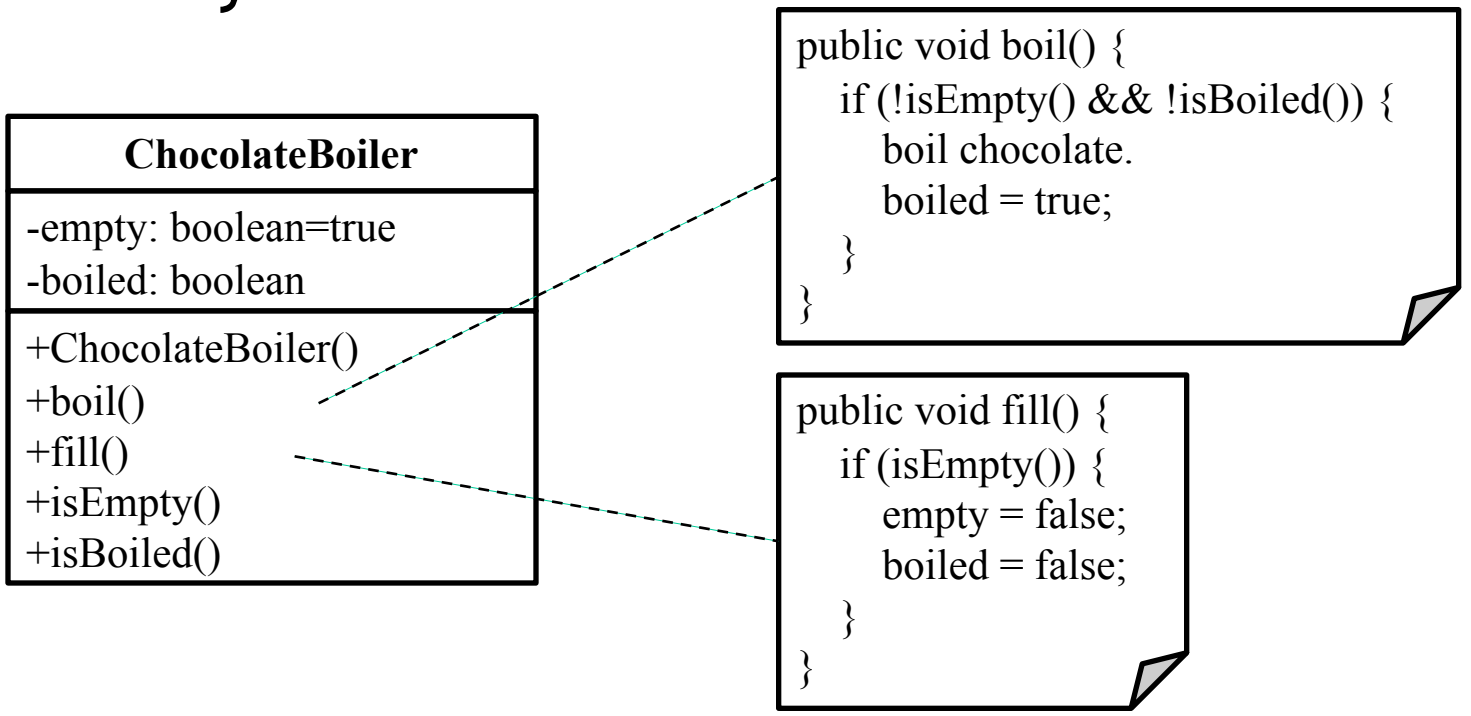
- A chocolate boiler is used to boil chocolate.

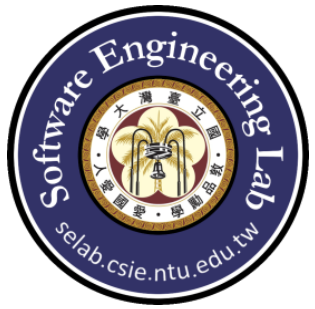




Requirements Statements$_2$

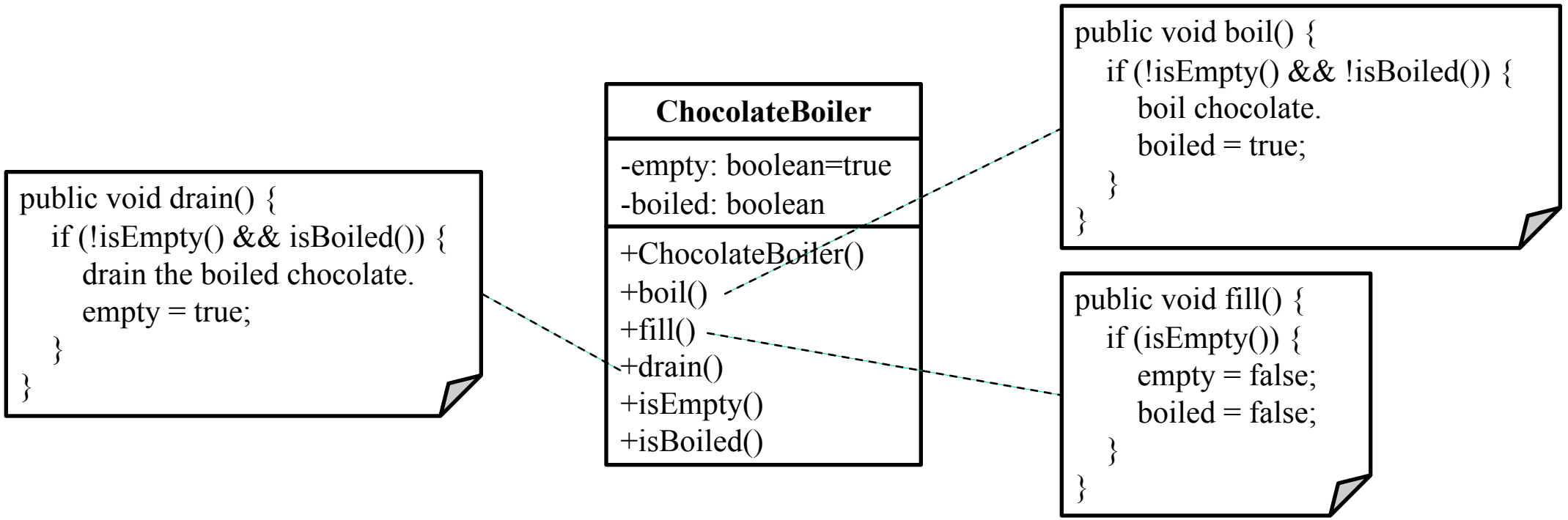
- Before boiling chocolate with the boiler, you have to make sure that the boiler is now empty and then fill chocolate in. Besides, you can't boil chocolate again while the chocolate has already been boiled.





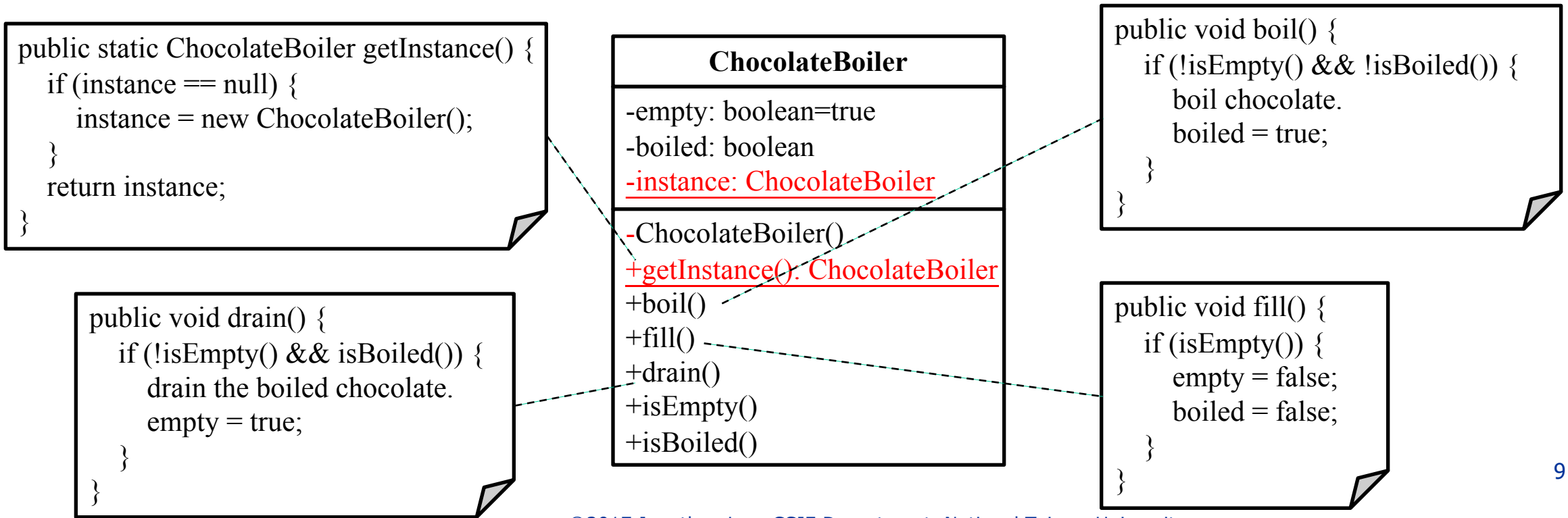
Requirements Statements$_3$

- After boiling, drain out the boiled chocolate and make the boiler empty again.



Requirements Statements$_4$

- In order to prevent some unexpected situation, it is not allowed to have multiple instances of the chocolate boiler in the system.



Initial Design

```
public static ChocolateBoiler getInstance() {
    if (instance == null) {
        instance = new ChocolateBoiler();
    }
    return instance;
}
```

ChocolateBoiler

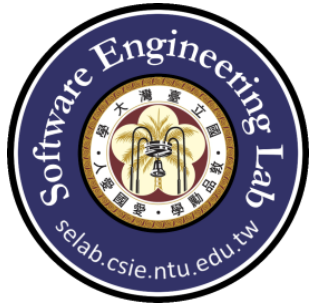
-empty: boolean=true
-boiled: boolean
-instance: ChocolateBoiler

-ChocolateBoiler()
+getInstance(): ChocolateBoiler
+boil()
+fill()
+drain()
+isEmpty()
+isBoiled()

```
public void boil() {
    if (!isEmpty() && !isBoiled()) {
        boil chocolate.
        boiled = true;
    }
}
```

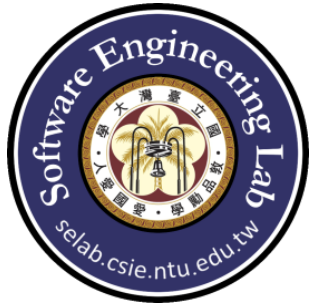
```
public void drain() {
    if (!isEmpty() && !isBoiled()) {
        drain the boiled chocolate.
        empty = true;
    }
}
```

```
public void fill() {
    if (isEmpty()) {
        empty = false;
        boiled = false;
    }
}
```



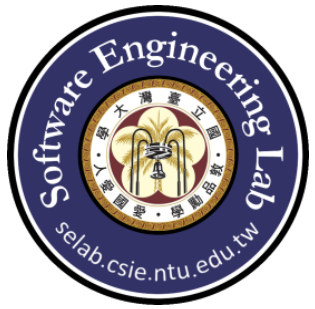
Recurrent Problem

- It's important for some classes to have exactly one instance and ensure that the instance is easily accessible.
- A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

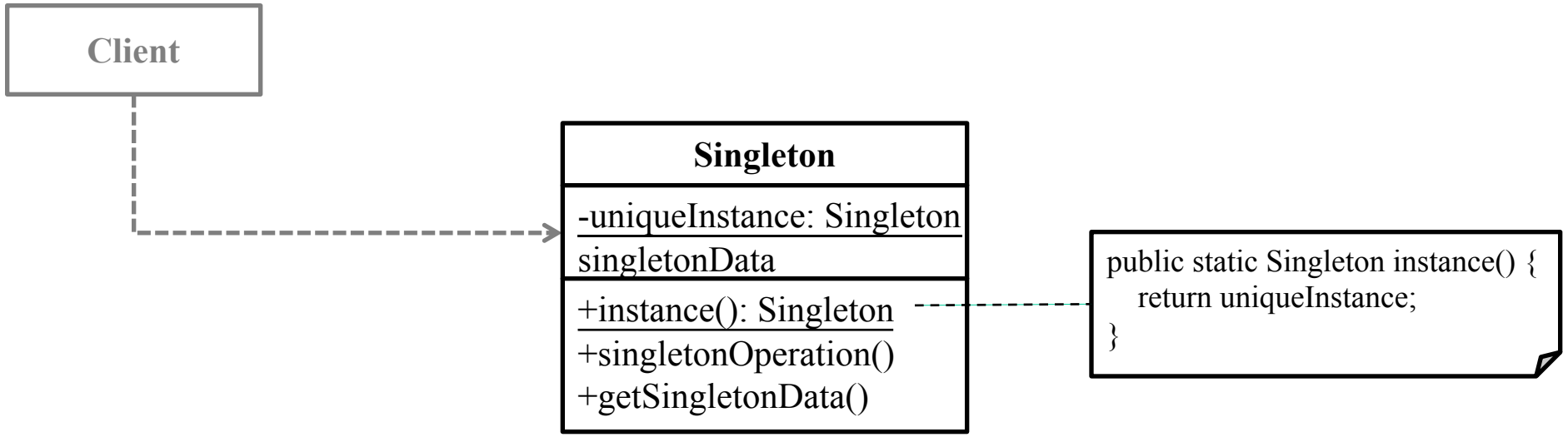


Intent

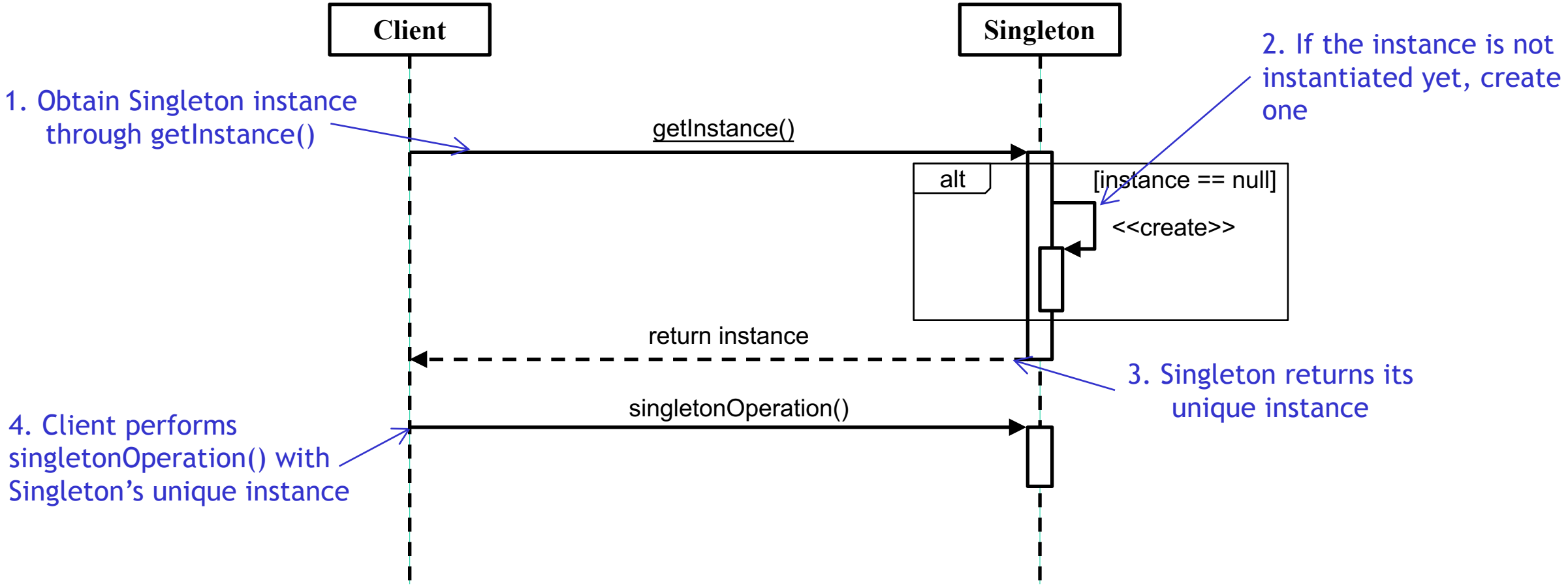
- Ensure a class only has one instance, and provide a global point of access to it.



Singleton Structure$_1$



Singleton Structure$_2$

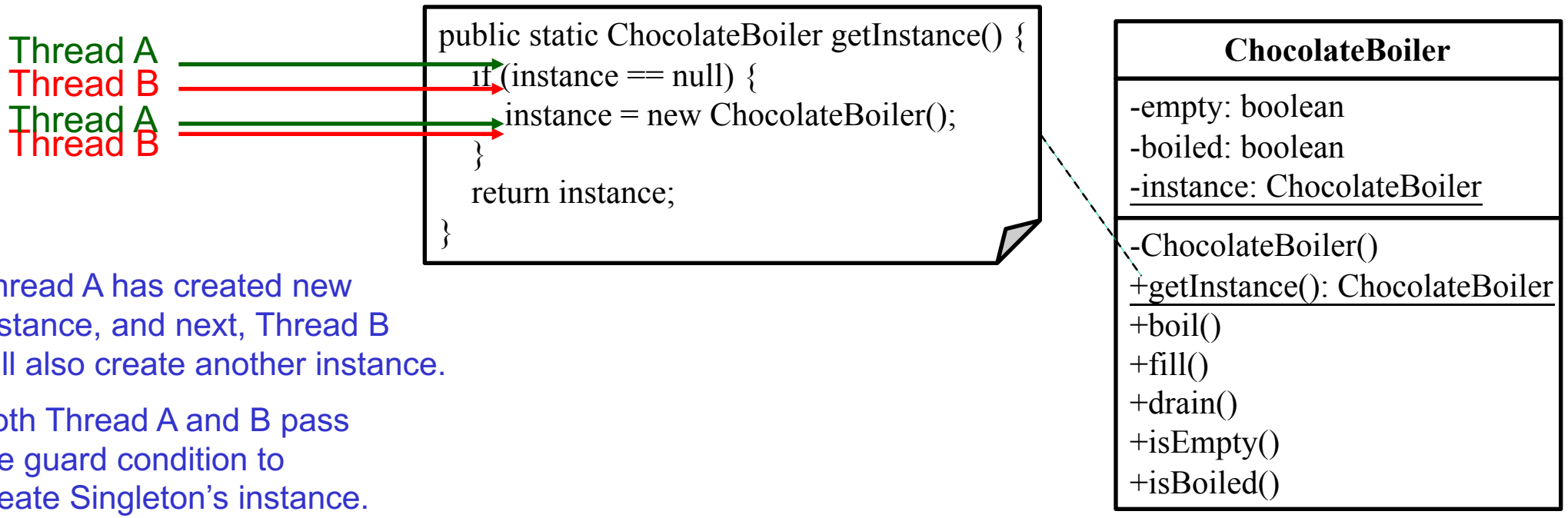


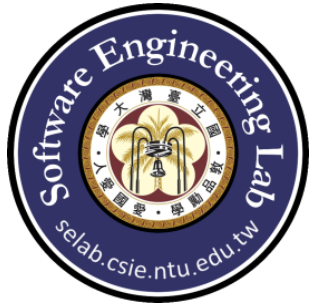
Singleton Structure$_3$

	Instantiation	Use	Termination
Singleton	Singleton creates itself and make sure there is only one instance.	Client gets the unique instance of Singleton from Singleton, and performs its operation(s).	Don't Care

Singleton with Multi-threading Issues$_1$

- **Issue:** In a multi-threading situation, if more than one thread request Singleton's instance, it may result in multiple instances which violates Singleton's intent.



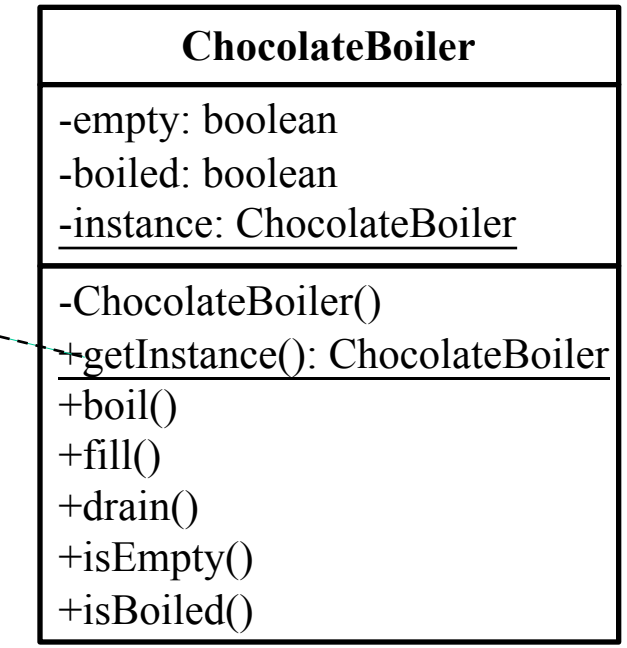


Singleton with Multi-threading Issues$_2$

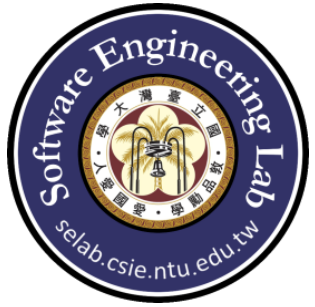
- **Solution 1:** Synchronizing the getInstance() method

This keyword in Java permits only one thread to enter the method at a time.

```
public static synchronized ChocolateBoiler getInstance() {
    if (instance == null) {
        instance = new ChocolateBoiler();
    }
    return instance;
}
```



However, if the instance has been created already, it doesn't make sense to allow only one thread to obtain the existing instance at the a time.

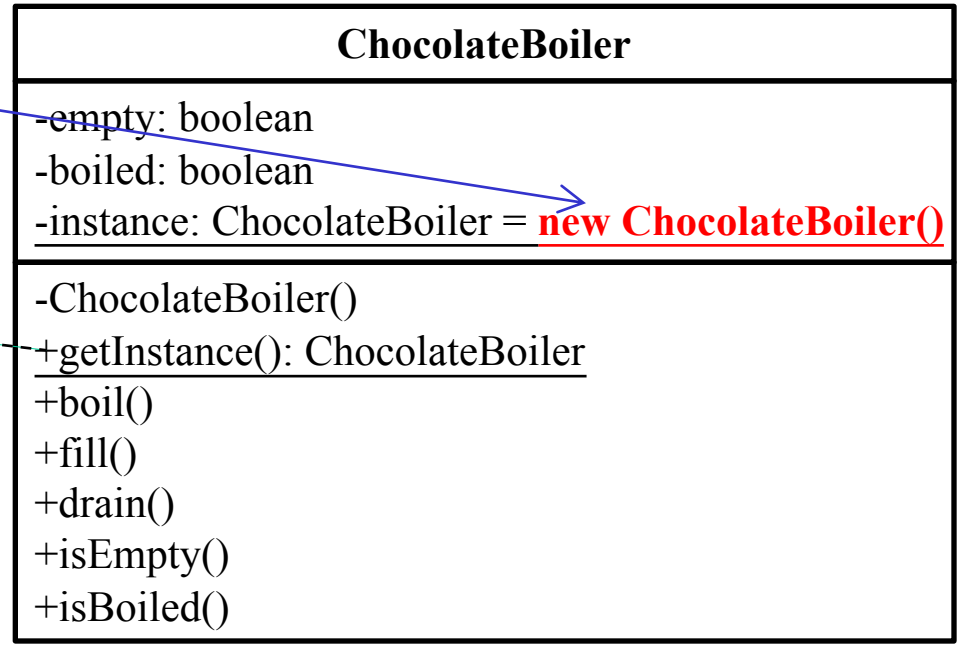


Singleton with Multi-threading Issues$_3$

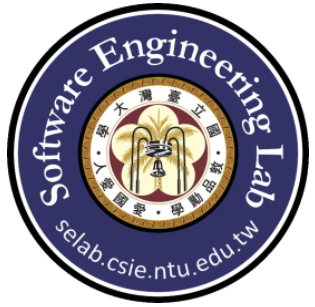
Solution 2: Eager creation

The instance is created when loading the ChocolateBoiler class, so the getInstance() method just return the early created instance to client.

```
public static ChocolateBoiler getInstance() {
    return instance;
}
```



It is a tradeoff to apply this approach if a Singleton instance occupies a huge amount of memory and is not needed so often.



Singleton with Multi-threading Issues$_4$

- **Solution 3:** Synchronizing the getInstance() method with double-checked locking

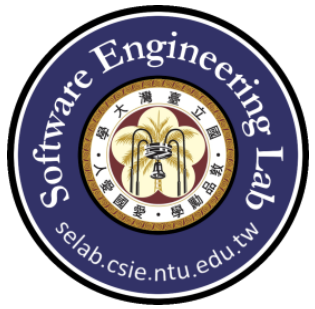
This statement in Java permits only one thread to enter this block at a time. If the instance is not created yet, the critical creation block allows only one thread to enter.

If the instance has been created, just return the existing instance without any synchronization.

Check nullity again in order to prevent creating instance more than once.

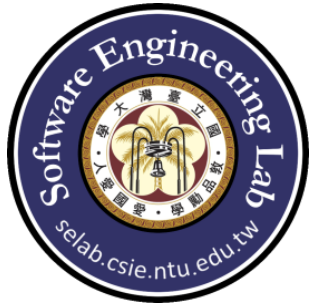
```
public static ChocolateBoiler getInstance() {
    if (instance == null) {
        synchronized (ChocolateBoiler.class) {
            if (instance == null) {
                instance = new ChocolateBoiler();
            }
        }
    }
    return instance;
}
```

ChocolateBoiler
-empty: boolean -boiled: boolean -instance: ChocolateBoiler
-ChocolateBoiler() +getInstance(): ChocolateBoiler +boil() +fill() +drain() +isEmpty() +isBoiled()



ChocolateBoiler

```
public class ChocolateBoiler {
    private boolean empty = true;
ackage-private more... (⌘F1)  oiled = false;
    private static ChocolateBoiler instance = new ChocolateBoiler();

    private ChocolateBoiler(){

    }

    public static ChocolateBoiler getInstance(){
        return instance;
    }

    public void boil() {
        if( !isEmpty() && !isBoiled() ){
            System.out.println("Boil chocolate");
            boiled = true;
        }
    }

    public void fill(){
        if(isEmpty()){
            System.out.println("Fill chocolate");
            empty = false;
            boiled = false;
        }
    }

    public void drain(){
        if(!isEmpty() && isBoiled()){
            System.out.println("Drain the boiled chocolate");
            empty = true;
        }
    }

    public boolean isEmpty() { return empty; }

    public boolean isBoiled() { return boiled; }
```



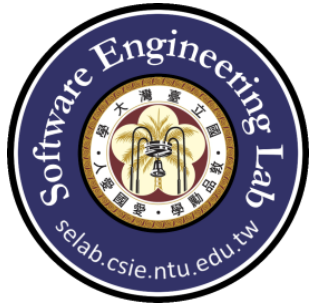
Input / Output

Input:

```
[Boil_chocolate_step]

...
```

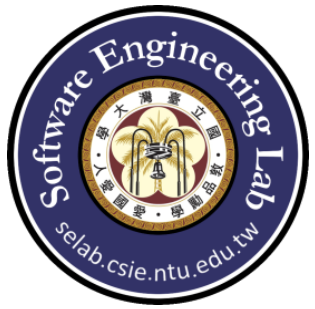
Output:

```
//if [Boil_chocolate_step] is Fill

Fill chocolate



//if [Boil_chocolate_step] is Boil

Boil chocolate



//if [Boil_chocolate_step] is Drain

Drain the boiled chocolate



...
```



Test cases

- TestCase 1: Basic correct three operation
- TestCase 2: Random 100 operation



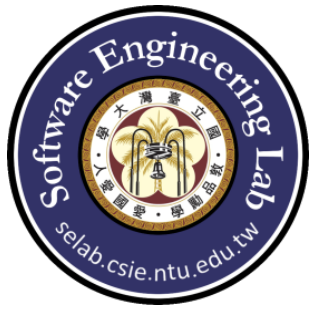
Test case1

Sample1.in

```
1 Fill
2 Boil
3 Drain
```

Sample1.out

```
1 Fill chocolate
2 Boil chocolate
3 Drain the boiled chocolate
```



Test case2

Sample2.in

```
Fill
Fill
Drain
Boil
Fill
Fill
Boil
Drain
Fill
Drain
Fill
Fill
Drain
Boil
Boil
Drain
Drain
Boil
Drain
Drain
Fill
Drain
Drain
Boil
Boil
Fill
Boil
Boil
Boil
Drain
Boil
Boil
Drain
Fill
Drain
Fill
Fill
Boil
Fill
Drain
Fill
Boil
Drain
Drain
Fill
Drain
Drain
Drain
Drain
Drain
Drain
Boil
Boil
Drain
Drain
Drain
Fill
Drain
Fill
Fill
Boil
Drain
Drain
Drain
Fill
Fill
Drain
Fill
Drain
Drain
Drain
Drain
Drain
Boil
Drain
Fill
Fill
Fill
Drain
Fill
Boil
Fill
Drain
Boil
Boil
Fill
Boil
Boil
Boil
Drain
Drain
Drain
```

Sample2.out

```
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
Fill chocolate
Boil chocolate
Drain the boiled chocolate
```