类实现正确性的论证模板及实例v2017

*说明1：此模板按照课堂所讲授内容，从抽象对象的正确实现、表示不变式正确保持和方法正确实现三个方面进行类实现正确性论证。论证是一种结构化的逻辑推理，类实现正确性论证的核心是论述代码实现如何满足所设计的规格。*

*说明2：此模板为配合教学需要而设计的模板，老师保留该模板的修正权利。任何使用该模板从事商业项目的行为，都应该事先征得老师的同意。该模板部分参考了教材中的论证思路，但做了相应扩展和细化。*

*说明3：即便按照此模板进行正确性论证，也并不能100%确保代码的安全正确，因为规格设计本身可能有错。*

1. 抽象对象得到了有效实现论证

论证类的overview明确了抽象对象，且类的rep能够通过抽象函数映射到相应的抽象对象。在有继承层次的情况下，要注意区分子类抽象函数与父类抽象函数的关系，并能够有效重用父类的相应论证结果。

*注：这一步论证较为简单。前提是必须在overview部分明确交代类的抽象对象（即一个类用来管理和处理什么样的数据），然后使用抽象函数就可获得论证。*

1. 对象有效性论证

论证在任意时刻，一个类的任意对象都能保证repOK为真，即任意一个方法的执行都不会导致rep为假。如果类的属性声明为public，则无法进行有效论证。在有继承层次的情况下，如果父类所有的rep都通过private进行保护，则无需进行额外论证，否则（即声明为protected）必须针对每个对父类rep状态进行修改的方法进行论证，证明不会导致父类对象的repOK为假。

*注：首先假设repOK的实现是正确的。在方法论证部分对repOK的正确性进行论证。*

1. 针对构造方法，论证初始态对象的repOK为真。

TokenMachine提供了一个构造方法，TokenMachine ()，它初始化全部的rep，repOK的运行结果显然返回true。

1. 逐个论证每个对象状态更改方法的执行都不会导致repOK的返回值为false。

TokenMachine提供了五个状体更新方法：requestService、plugServiceWindow、unPlugServiceWindow、serveNext、tokenSatisfied，下面逐个进行论证。

* 假设requestService(ServiceType type)方法开始执行时，repOK为true。

1. requestService方法首先根据输入的服务类型，使用 queue和num来引用(waitingCreditServiceTokens, num\_creditSW) / (waitingSavingServiceTokens, num\_savingSW)，显然不改变repOK的取值；
2. 然后判断如果当前有与请求服务类型相匹配的窗口提供服务，则通过库方法size获得等待队列的长度，token编号计数开始加1，使用新的编号计数count、服务类型type和队列中的等待请求数目waits来创建一个新的token，并加入到相应的等待队列中。由于Token类实现正确，确保其初始状态满足要求，且这个新token的编号一定大于等待队列中所有其他token的编号，且不同于所有已存在的token，因此这个新加入的新token对象不会导致repOK为false；
3. 返回所创建的token，结束方法执行，因此repOK依然为真。
4. 因此，该方法的执行不会导致repOK为假，不违背表示不变式。

* 假设serveNext(ServiceWindow win)方法开始执行时，repOK为true。

1. 首先如果win不是空闲状态，则表明无法来服务下一个请求，方法结束，未改变任何表示对象，不会导致repOK为假；
2. 根据win提供的服务类别获得对相应等待队列的引用，不改变对象状态；
3. 接下来迭代访问servingTokens中所有对象。对于任意服务窗口为win的token而言，如果该token正在接受服务，因为ServiceWindow类实现正确，那么win的状态就必然是serving，在第1步论证中就会结束执行；如果该token已经被服务完毕，那么它就应该从servingTokens中被移除，而出现在servedServiceTokens中了，显然也不是这种情况；此时，只有一种情况，即该token丢失了，即拿着该token的客户离开了服务场所，因此从servingTokens中移除相应的token。由于不改变servedServiceTokens，不会导致repOK为假；
4. 从与win服务类别相匹配的等待队列中取出头请求token，如果token不为空则加入servingTokens中，标识token在win接受服务，并通知去win接受服务。队列Queue的性质确保了队列头token的编号小于队列中所有其他token的编号，该token从等待队列移动到servingTokens，确保servingTokens中的任意token编号必然小于等待队列中的编号；因此repOK依然为true；
5. 方法执行结束，因此该方法执行不会导致repOK为假。

…

1. 该类的其他几个方法的执行皆不改变对象状态，因此这些方法执行前和执行后的repOK都为true。
2. 综上，对该类任意对象的任意调用都不会改变其repOK为true的特性。因此该类任意对象始终保持对象有效性。
3. 方法实现正确性论证

目标：在方法的前置条件和后置条件没有逻辑矛盾的前提下，根据前置条件和后置条件分析整理出完整的{<effect Y> with <input partition X>}，然后对照方法代码实现来论证给定每个<partition X>，方法执行结果都满足<effect Y>。

*注意在继承层次下，子类方法如果是重载了父类方法，则需要论证：父类方法的前置条件蕴含子类方法的前置条件、子类方法的后置条件蕴含父类方法的后置条件。*

1. serveNext(ServiceWindow win):

/\*@Requires: win!=null && win.status == idle

@Modifies: this

@Effects: \all int i; 0<=i<servingTokens.size; servingTokens[i].winID != win.id;

(win.type == creditService) ==> \invoked(waitingCreditServiceTokens.poll) && (waitingCreditServiceTokens.poll != null ==> servingTokens.contains(waitingCreditServiceTokens.poll) &&

\invoked(win.Serve(waitingCreditServiceTokens.poll)));

(win.type == savingService) ==> \invoked (waitingSavingServiceTokens.poll) && (waitingSavingServiceTokens.poll != null ==> servingTokens.contains(waitingSavingServiceTokens.poll) &&

\invoked(win.Serve(waitingSavingServiceTokens.poll)));

\*/

根据上述过程规格，获得如下的划分：

<do nothing> with <win.status != *idle*>

<no waiting token will be picked, and all lost tokens will be removed from the serving token list> with <win.status == idle, no matched waiting token and there are *n* lost tokens>

<first matched token will be polled, added into serving token list, and win.Serve will be invoked, and all the lost tokens will be removed from the serving token list> with <win.status == idle, there are matched waiting token(s), and there are *n* lost tokens>

* 方法首先检查确认输入win的状态，如果不是idle，则直接返回结束，满足<do nothing> with <not idle win>
* win状态为idle，通过方法检查。根据win的服务类型获得对相应等待队列的引用queue，通过对servingTokens的迭代访问，可以取得所有n个服务窗口为当前win的token，这些token皆为lost token，直接从servingTokens中删除。此时由于等待队列中没有与win服务类型相匹配的token，因此queue.poll()操作取得的token将为null，执行直接结束。在此过程中只修改了this所管理的servingTokens, 未对win和其他对象进行修改。因此，满足<no waiting token will be picked, and all lost tokens will be removed from serving token list> with < win.status == idle, no matched waiting token and there are *n* lost tokens>
* win状态为idle，通过方法检查。根据win的服务类型获得对相应等待队列的引用queue，通过对servingTokens的迭代访问，可以取得所有n个服务窗口为当前win的token，这些token皆为lost token，直接从servingTokens中删除。此时等待队列中有与win服务类型相匹配的若干token，queue.poll()将取出其中等待时间最长的token，显然不为null，接下来token被加入到servingTokens队列中，并设置token的服务窗口为win，通过调用win.Serve来服务该token。显然，token满足Serve方法的前置条件要求，Serve方法对token的处理将满足其后置条件。如果该token没有lost，则将得到正常处理，否则在win下一次调用ServeNext时，该lost token会被识别并移除。在此过程中只修改了this所管理的servingTokens, 等待队列和选出的token，未对win和其他对象进行修改。因此，处理满足<first matched token will be polled, added into serving token list, and win.Serve will be invoked, and all the lost tokens will be removed from the serving token list> with < win.status == idle, there are matched waiting token(s), and there are *n* lost tokens>。

1. tokenSatisfied(Token token)

/\*@Requires: token != null && token.staus ==serving.

@Modifies: this

@Effects: \old(servingTokens).contains(token) ==> \invoked(token.Satisfied) && servedServiceTokens.contains(token) &&

!servingTokens.contains(token)\*/

根据上述过程规格，获得如下的划分：

<do nothing> with <servingTokens.contains(token) == false>

<!servingTokens.contains(token) && servedServiceTokens.contains()token && \invoked(token.Satisfied)> with <servingTokens.contains(token)>

* 方法首先检查确认输入输入的token是否在servingTokens中，如果不在，执行结束，this状态没有改变。因此满足<do nothing> with <servingTokens.contains(token) == false>。
* 如果token出现在servingTokens中，方法首先调用token.Satisfied()。因为token状态为serving，根据token.Satisfied的规格，该方法会把token状态改变为served。然后把token从servingTokens中移出，加入到servedServiceTokens中。因此满足<!servingTokens.contains(token) && servedServiceTokens.contains()token && \invoked(token.Satisfied)> with <servingTokens.contains(token)>。

综上所述，所有方法的实现都满足规格。从而可以推断，TokenMachine的实现是正确的，即满足其规格要求。