前言：

本文对一个简化了的版本控制软件做了面向对象的分析与设计，该版本控制软件类似于Git，并对功能进行了简化，我们将其命名为Toygit。所使用的建模工具为UMLet。

本文的主要工作包括：（1）首先对Toygit进行了需求分析，生成了需求模型，即用况图。（2）在需求模型的指导下，对系统进行了初步的分析，得到了类图与部分辅助图模型（3）针对问题域与数据接口部分对OOA阶段的的基本模型进行了细化与重新设计，得到了OOD模型以及部分更为具体的辅助模型。

背景：

在设计、开发、写作等过程中，我们常常需要在每个阶段都对现已完成的工作进行标记、备份，以便在工程出现问题或因为某些目的需要追踪所做的更改时，可以回滚到之前的状态。版本控制软件就是一种方便人们进行标记、备份的软件，它可以记录一个或若干文件内容变化，并以一种简洁易用的方式供用户查阅特定版本修订情况。

Git是当前比较流行的开源版本控制软件，本文的主要工作是对一个类似于Git，功能较为简单的版本控制软件进行面向对象的分析与设计。我们在下文中，将该软件命名为Toygit。

名词解释：

仓库：每个被版本控制软件所管理的项目，具体表现为一个存放项目的文件夹。

版本：用户对项目内容的每一次提交。

分支：每个项目可以朝不同的路径演化，不同路径互不影响，每个路径即为一个分支。

当前版本：所在分支的最近一次提交。

暂存区：存放那些作为下一版本的文件的缓冲区，提交命令使得暂存区中的所有文件成为一个新版本

功能介绍：

创建仓库：每个仓库相当于版本控制软件所管理的一个项目，表现为一个存放项目的文件夹，每个用户只能访问自己的仓库。

提交版本（commit）：将项目中指定的内容作为一次版本进行提交，每个版本有唯一的id号。

切换分支(checkout <branch\_name|commit\_id>)：给定分支名，将项目切换到指定分支，并将工作目录内容设置为该分支的当前版本。当给定分支名为版本id号时，切换到该id对应的版本。

回滚版本(reset <commit\_id>)：在当前分支下，将项目回滚到具体的某一个版本，并将该状态作为当前分支的当前状态。

存入暂存区(add <file>)：将指定文件或文件夹添加到暂存区。

移出暂存区(remove <file>)：将指定文件或文件夹移出暂存区，同时从工作目录中删除。

重置暂存区(reset <file>)：将暂存区重置成当前版本的初始状态，亦可指定只重置暂存区中具体的文件或文件夹。

恢复文件(checkout <file>)：将文件或文件夹恢复成暂存区中的内容。

创建分支(branch <branch\_name>)：给定分支名，创建一个新的分支，该分支的当前版本为创建该分支时，所在分支的当前版本。系统默认分支为master。

删除分支(branch –d <branch\_name>)：删除某一分支。

添加标记(tag commit\_id)：给定版本id号与标记名，将该名字作为对应版本的标记。标记相当于版本的引用。

删除标记(remove <tag\_name>)：删除某一标记。

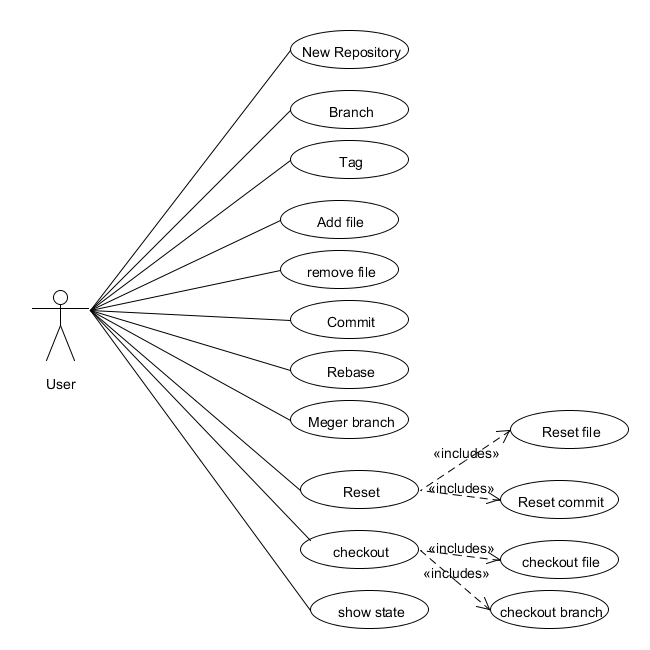
分支合并：给定分支名，将当前分支与指定分支的当前版本进行合并，合并后的版本作为一次新的提交，并将该提交设为当前分支的当前版本。如果合并过程存在冲突，则提示用户手动解决冲突，冲突解决后继续合并。

分支衍合：给定分支名，找到当前分支与指定的分支的最近公共祖先版本，为当前分支从该祖先版本开始的每次提交生成一个补丁，将所得的一系列补丁打在指定分支的当前版本上，最后生成一个新的版本，将其作为当前分支的当前版本。如果衍合过程存在冲突，则提示用户手动解决冲突，冲突解决后继续衍合。

显示当前目录状态：显示工作目录与暂存区中不相符的文件。

**OOA**

用况图：



用况：

**commit [description]：提交版本**

if(用户输入强制commit命令) {

if(用户未输入description) {

提示用户输入description;

输入description

}

给暂存区生成快照作为一次新版本；

记录并输出当前commit信息，包括commit\_id, author, data, description

设置当前分支的当前提交为该新版本

}

if(用户输入commit命令) {

if (存在新增、已修改，但尚未暂存的文件 || 存在已删除，但未移出暂存区的文件){

提示用户add或rm后，再执行commit命令;

结束用况;

}

if(用户未输入description) {

提示用户输入description;

输入description

}

给暂存区生成快照作为一次新版本；

记录并输出当前commit信息，包括commit\_id, author, data, description

将该版本设置为当前分支的最新版本

}

**add <filename>：存入暂存区**

用户输入add <filename>命令

if (<filename>是普通文件) {

用<filename>文件替换暂存区中的<filename>文件

}

if (<filename>是目录文件) {

递归地用<filename>下所有文件替换暂存区中对应的文件

}

}

**reset命令：回滚当前版本，恢复暂存区**

用户输入reset <filepath|commit\_id|tag>

if(输入的是commit\_id) {

当前分支回滚到commit\_id所对应的版本

暂存区的内容重置为commit\_id所对应版本的文件内容

}

else if（输入的是tag）{

当前分支回滚到tag所对应的版本

暂存区的内容重置为tag所对应版本的文件内容

}

else if(输入的filepath) {

if(如果<filepath>是普通文件) {

将暂存区中的<filepath>文件恢复为当前版本的原始状态

}

if(如果<filepath>是目录文件){

递归地将暂存区中，<filepath>下的所有文件恢复为当前版本的原始状态

}

}

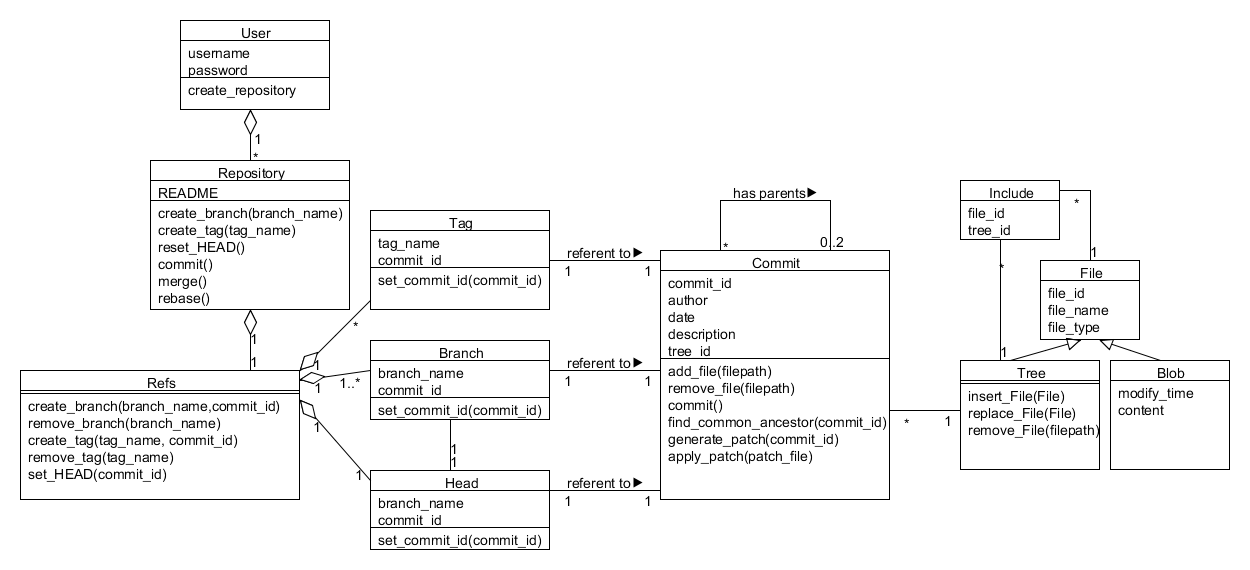
else {

暂存区的内容重置为当前版本的文件内容

}

}

**基本模型——类图**



对项目的管理，实际上是对文件与文件夹的管理，一个项目本质就是一个文件夹，文件夹下面含有若干文件与子文件夹。由于文件的类型是多种多样的，因此将其抽象成为Blob类型，即binary large object类型，而文件夹将其抽象成为Tree类型。借鉴操作系统中文件系统的实现，将Tree与Blob都抽象成File类型，即Tree和Blob继承File。在逻辑上，一个Tree可以包含任意多个File（即Tree或Blob），然而跟传统的文件系统不一样，每个File可以被多个Tree所包含，具体考虑如下：对于相邻的版本之间，往往只修改了少量文件，对于未做修改的文件，我们不希望在每个版本都为其创建一个新的对象，而通过引用的方式进行复用。因而，File与Tree之间是多对多的关联关系，通过中间类Include将其转化为一对多关联。

对于版本，将其定义成为Commit类，每个Commit引用一个处于根目录的Tree对象，即可通过版本对象Commit获得对应版本的所有文件信息。同样基于复用未做修改的对象的考虑，每个Tree对象可以被多个Commit对象引用。

版本的演进可以用Commit对象之间的父子关系抽象，上一次提交的版本是这一次提交版本的父亲，由于分支功能的需求，一个Commit对象可以有任意多个儿子，由于分支合并的功能需求，每个Commit对象可以有0到2个父亲（分别对应首次提交的版本，线性提交的版本，分支合并后的版本），由于是自己到自己的多对多（我们将2也视为多），因此引用中间类的策略不再奏效，除非在Commit与中间类之间需要两条（单向）关联线，这只会使模型变得杂乱，因此我们保持Commit自身到自身的多对多关系，将这个问题留待OOD阶段解决。

基于Commit在逻辑上的链式结构，对于分支的功能，实际上只需让每个分支记录其当前版本（Commit）的id即可，这样，分支之间的切换就变成了版本的切换。

由于每次进入系统时，都需要知道正在哪一个分支工作，或对于临时分支而言，当前版本是哪一个，因此需要一个记录当前分支或当前版本的类，即HEAD类。

对于给版本做标记，也只需一个能够记录被标记Commit的id的类即可，即Tag类。

Branch，HEAD，Tag类本质是都属于引用类，所以为它们设置一个共同的父类——引用类，每个仓库拥有多个引用类，然而这样的模型无法描述每个仓库必须至少拥有一个Branch类（默认的master分支）以及有且仅有一个HEAD。因此，将继承转换成了聚合，一个引用类Refs拥有若干Branch，HEAD和Tag类，一个仓库Repository拥有一个Refs类。