**软件设计与体系结构 参考答案**

1. 单项选择题（每小题2分，共20分）

1~5：CCDDA 6~10：ABBAA

1. 填空题（每空1分，共10分）
2. 依赖倒转
3. 模型、控制器
4. 访问者
5. 骨架、子类
6. 装饰器
7. 职责链
8. Command、Receiver
9. 简答题（每小题3分，共15分）
10. 就一个类而言，应该仅有一个引起它变化的原因。
11. A.高层模块不应该依赖于低层模块。两个都应该依赖于抽象。

B.抽象不应该依赖于细节，细节应该依赖于抽象。

1. 尽量使用合成/聚合，尽量不要使用类继承。
2. 定义一个操作的算法骨架，而将一些步骤延迟到子类中，模版方法使得子类可以不改变一个算法的接口即可重定义改算法的某些特定步骤。
3. 将对象组合成树形结构以表示“整体-部分”的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。
4. 程序填空题（共25分）
5. (2分)

Tree createTree();

Floor createFloor();

(2)（6分）

@Override

public Tree createTree() {

return new WinterTree();

}

@Override

public Floor createFloor() {

return new WinterFloor();

}

(3)（2分）

GameDialog gameDialog=new GameDialog();

gameDialog.setFactory(new WinterFactory());

gameDialog.init();

2、

1. (5分)

private static GameScore instance=null;

private GameScore() {

}

public static GameScore getInstance() {

if (instance == null) {

instance = new GameScore();

}

return instance ;

}

1. (10分)

private List<Observer> observersList=new ArrayList<Observer>();

@Override

public void registerObserver(Observer o) {

observersList.add(o);

}

@Override

public void removeObserver(Observer o) {

observersList.remove(o);

}

@Override

public void notifyObserver() {

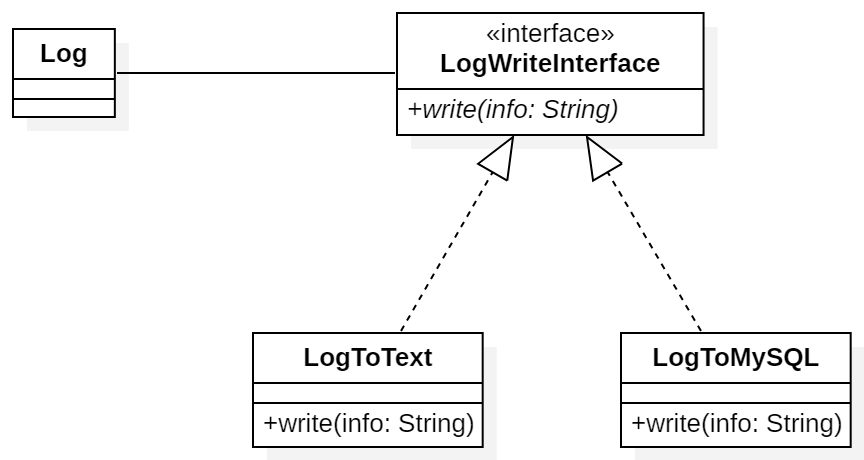
for (Observer o : observersList){

o.update(this.gameTime,this.homeScore,this.visitorScore);

}

}

1. 应用题（共30分）
2. UML类图（5分）：



伪代码（10分）：

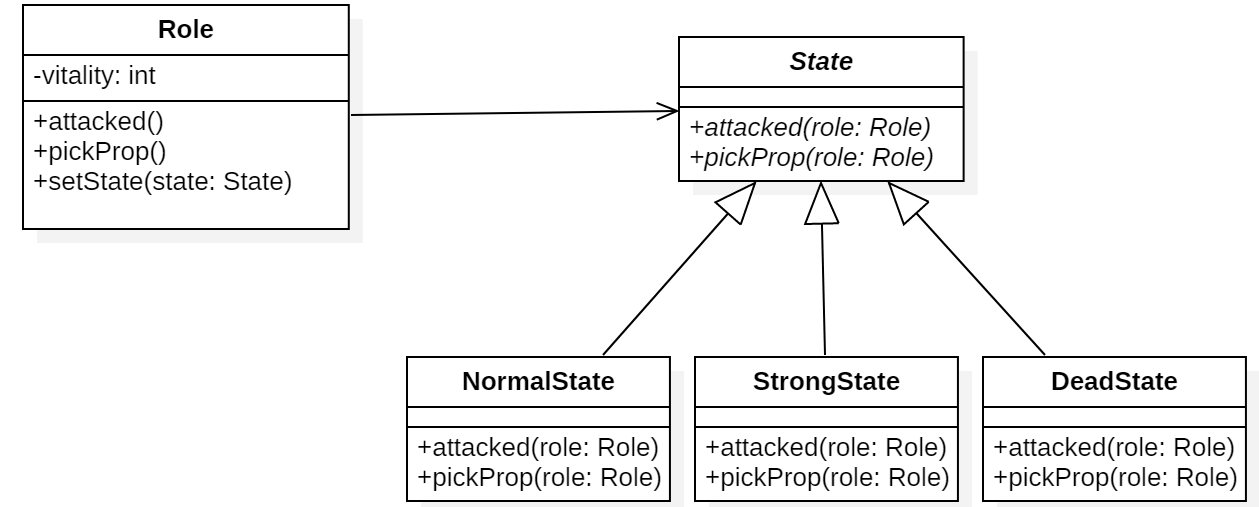
**public class** Log {  
 **private** LogWriteInterface **logWriter**;  
  
 **public void** setLogWriter(LogWriteInterface logWriter) {  
 **this**.**logWriter** = logWriter;  
 }  
 **public void** writeLog(String info){  
 **logWriter**.wirte(info);  
 }  
}

**public interface** LogWriteInterface {  
 **void** wirte(String info);  
}

**public class** LogToText **implements** LogWriteInterface{  
 @Override  
 **public void** wirte(String info) {  
 System.***out***.println(**"日志信息写入文本文件"**);  
 }  
}

**public class** LogToMySQL **implements** LogWriteInterface {  
 @Override  
 **public void** wirte(String info) {  
 System.***out***.println(**"日志信息写入MySQL数据库"**);  
 }  
}

2、UML类图（5分）



伪代码（10分）：

**public class** Role {  
 **private int vitality**=5;  
  
 **private** State **state**;  
 **private** NormalState **normalState**;  
 **private** StrongState **strongState**;  
 **private** DeadState **deadState**;  
  
 **public void** attacked(){  
 **state**.attacked(**this**);  
 }  
 **public void** pickProp(){  
 **state**.pickProp(**this**);  
 }  
 **public void** setState(State state) {  
 **this**.**state** = state;  
 }  
  
 **public int** getVitality() {  
 **return vitality**;  
 }  
 **public void** setVitality(**int** vitality) {  
 **this**.**vitality** = vitality;  
 }  
 **public** NormalState getNormalState() {  
 **return normalState**;  
 }  
 **public** StrongState getStrongState() {  
 **return strongState**;  
 }  
 **public** DeadState getDeadState() {  
 **return deadState**;  
 }  
}

**public abstract class** State {  
 **public abstract void** attacked(Role role);  
 **public abstract void** pickProp(Role role);  
}

**public class** NormalState **extends** State {  
 @Override  
 **public void** attacked(Role role) {  
 **int** vitality=role.getVitality();  
 vitality-=1;  
 role.setVitality(vitality);  
 **if**(vitality==0){  
 role.setState(role.getDeadState());  
 }  
 }  
 @Override  
 **public void** pickProp(Role role) {  
 role.setState(role.getStrongState());  
 }  
}

**public class** StrongState **extends** State {  
 @Override  
 **public void** attacked(Role role) {  
 role.setState(role.getNormalState());  
 }  
 @Override  
 **public void** pickProp(Role role) {  
 }  
}

**public class** DeadState **extends** State {  
 @Override  
 **public void** attacked(Role role) {  
 }  
 @Override  
 **public void** pickProp(Role role) {  
 }  
}