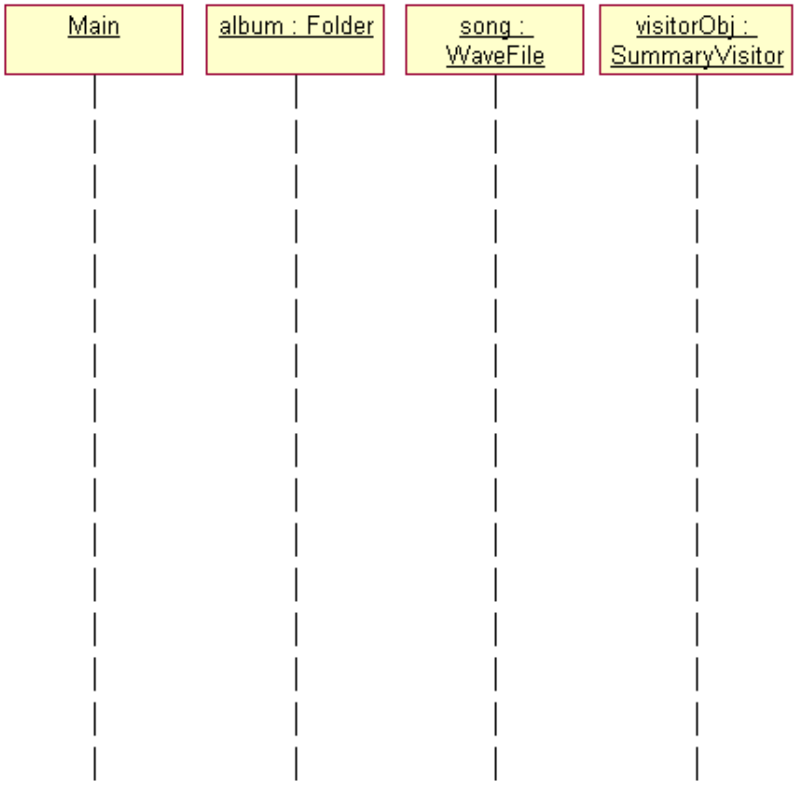
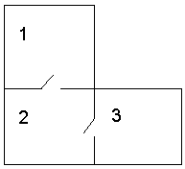
I lab 2 you where to use the adapter pattern to allow the WAV_IN and WAV_OUT classes to work with the Folder class. Describe how your solution to the lab was similar to the examples of the Adapter pattern in the book. Describe how your solution differed from the examples in the book.

Complete the sequence diagram below describing the call sequence used to create a summary of the album and song objects.



Rewrite the createAbstractFactoryMaze to create the following standard standard maze:



Recall the following:

```
class MazeFactory {
public:
  // ...
  virtual Maze* makeMaze() const;
  virtual Wall* makeWall() const;
  virtual Room* makeRoom(unsigned int rmNum) const;
  virtual Door* makeDoor(Room* rm1 = NULL, Room* rm2 = NULL) const;
  // ...
};
```

```
int main()
{
  MazeFactory factory;
  Maze* mazePtr = createAbstractFactoryMaze(factory);
  return EXIT_SUCCESS;
}

Maze* createAbstractFactoryMaze(MazeFactory& factory)
{
  Maze* theMaze = factory.makeMaze();
  Room* r1 = factory.makeRoom(1);
  Room* r2 = factory.makeRoom(2);
  Room* r3 = factory.makeRoom(3);
  Door* dr1 = factory.makeDoor(r1, r2);
  Door* dr2 = factory.makeDoor(r2, r3);

  theMaze->addRoom(r1);
  theMaze->addRoom(r2);
  theMaze->addRoom(r3);

  r1->setSide(Room::North, factory.makeWall());
  r1->setSide(Room::East, factory.makeWall());
  r1->setSide(Room::South, dr1);
  r1->setSide(Room::West, factory.makeWall());

  r2->setSide(Room::North, dr1);
  r2->setSide(Room::East, dr2);
  r2->setSide(Room::South, factory.makeWall());
  r2->setSide(Room::West, factory.makeWall());

  r3->setSide(Room::North, factory.makeWall());
  r3->setSide(Room::East, factory.makeWall());
  r3->setSide(Room::South, factory.makeWall());
  r3->setSide(Room::West, dr2);

  return theMaze;
}
```

Recall the AudioClipManager class developed in class:

```
class AudioClipManager {
public:
  static AudioClipManager* getInstance();
  void play(AudioClip* clip);
  void stop();
  static void cleanUp();
  // ...
private:
  AudioClipManager();
  ~AudioClipManager();
  static AudioClipManager* instance;
  static AudioClip* prevClip;  // Clip that may still be playing from
                               //  previous function call
};
```

Draw the sequence diagram for the code in the `for` loop below:

```
int main()
{
```

```
  AudioClip song(/*whatever it takes to actually create one of these*/);
  for(unsigned int i=0; i<3; ++i) {
    (AudioClipManager::getInstance())->play(&song);
  }
  return EXIT_SUCCESS;
};
```

```
AudioClip* AudioClipManager::prevClip = NULL;

AudioClipManager::AudioClipManager()
{
}

AudioClipManager::~AudioClipManager()
{
}

AudioClipManager* AudioClipManager::getInstance()
{
  if(instance==NULL) {
    instance = new AudioClipManager;
  }
  return instance;
}

void AudioClipManager::play(const AudioClip& clip)
{
  if(prevClip!=NULL) {
    stop();
  }
  prevClip = clip;
  // start playing
}

void AudioClipManager::stop()
{
  // stop playing
}

void AudioClipManager::cleanUp()
{
  if(instance) {
    delete instance;
  }
}
```

Explain how the **command** pattern can be used to facilitate “undo/redo” support. Discuss the two techniques used to implement this action.

Suppose the **State** pattern is used to implement a singly-linked list class. Indicate what classes associated with the State pattern should be created (along with member functions) in order to handle the following operations:

```
{
  SLList<int> lst;
  lst.push_front(1);
  lst.push_front(2);
  lst.push_front(3);
}
```

Draw a sequence diagram describing the sequence of events for the above code. You may assume that the state objects are already created (just tell me what they are).