# Yet More Maze, Yet More Design Patterns

CS151

Chris Pollett

Nov. 16, 2005.

# Outline

- More on Maze Game
- AbstractFactory Pattern
- Factory Method Pattern
- Prototype Pattern

# More on Maze Game

- Last day, we said the Maze Game was built out of classes:
  - MapSite-- extended by Door, Wall, Room. It is an interface which supports clone(), enter(Maze m), and draw.
  - Room -- has four MapSite's on it and each MapSite is associated with 1 or 2 rooms. A Room has a number, and has constants for ROOM_COLOR and PLAYER_COLOR. It has accessor methods: getRoomNumber, getLocation, isInRoom. It has mutators: setRoomNumber, setLocation, setInRoom, setSide(dir, site). The function enter(Maze m) allows one to set a room as the current room of the Maze m.
  - A Maze has 1 or more Room's
  - SimpleMazeGame is used to drive the whole game

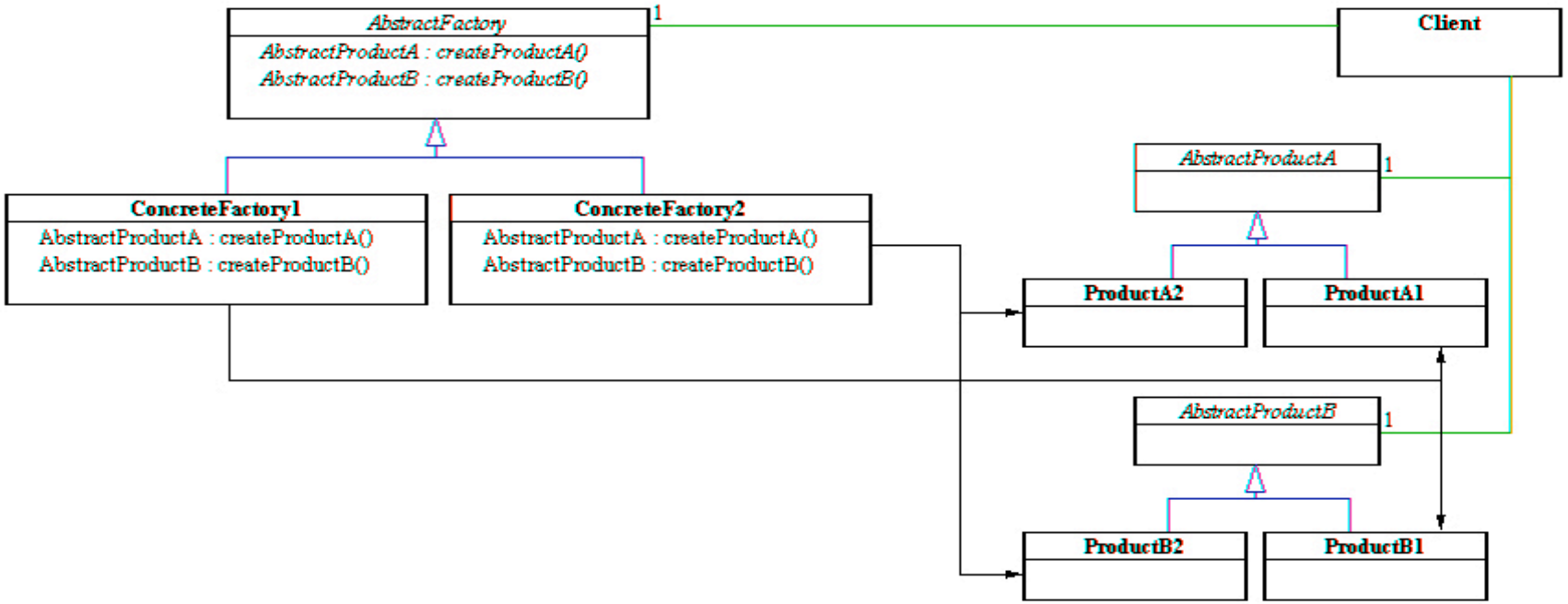
# Still more on Maze Game

- Door -- implements MapSite and has methods isOpen, setOpen, setRooms(r1, r2) getOrientation, setOrientation, enter(Maze maze) (works if open otherwise plays a ding sound)
- Wall -- implements MapSite. Trying to enter a Wall causes a ‘hurts’ sound to be played.
- The SimpleMazeGame has two methods createMaze() and createLargeMaze. The first creates a 1x2 maze, the latter a 3x3 maze.

# Adding Interest

- We want to make the game more interesting. So we want to support different themes.
- We will make Harry Potter theme by making a HarryPotterXXX class where XXX is a subclass of Door, Wall, or Room.
- Similarly, we will make SnowWhiteXXX where XXX is a subclass of Door, Wall, or Room.
- The theme affects things like what audio clip is played when switching rooms, the color of a Room, the color of a Walls,etc.
- Now to use these themes we could awkwardly add to SimpleMazeGame methods createHarryPotterMaze() createLargeHarryPotterMaze(), createSnowWhiteMaze(), createLargeSnowWhiteMaze().
- Instead we'll try to use a different design patterns, such as: abstract factory, factory method, prototype, and builder.

# AbstractFactory Pattern



# More on Abstract Factory

- We will create a class MazeFactory (our AbstractFactory) which supports makeMaze, makeWall, makeRoom, makeDoor.
- We create concrete subclasses HarryPotterMazeFactory and SnowWhiteMazeFactory, which create concrete Wall, Door, and Room objects of the given theme.
- Our client, which will be the Maze, only uses the MazeFactory interface to make Wall's, Door's, and Room's.
- We'll have a MazeGameAbstractFactory which can build a complete maze using a supplied MazeFactory's methods. That is, MazeGameAbstractFactory will have the two methods: createMaze(MazeFactory fac) or createLargeMaze(MazeFactory fac)

# Factory Method Pattern

- Another way to solve this same problem is to use the Factory Method pattern that we've discussed earlier.
- We could make a class MazeGameCreator which has methods createMaze, createLargeMaze. makeMaze, makeWall, makeRoom, makeDoor.
- Then we could subclass this into HarryPotterMazeGameCreator and only override makeWall, makeDoor, makeRoom.
- Thus, the actual rooms in the Maze are determined by the base class but how these rooms look is determined by the subclass.
- Similarly, we could create a subclass SnowWhiteMazeGameCreator.
- What theme a game uses then depends on which MazeGameCreator is used.

# Prototype Pattern

- One drawback of the factory patterns is that we need to subclass both products and factories when we create a new theme.
- If there are many themes then this can become unflexible.
- Instead, can use a prototype pattern.
- In the pattern, we have an abstract class Prototype (Maze, Wall, Door) which is Cloneable. Create concrete subclasses (HarryPotterWall). A Client (MazeProtoTypeFactory) creates new instances by cloning the particular concrete prototypes that have been placed on it.
- MazeGameAbstractFactory sets the prototypes on this factory then creates the maze.