Review: The Maze Example

- Building a maze for a computer game
- A Room knows its neighbours through sides
  - another room, a wall, a door
- A Maze is a set of rooms
  - It does not keep references to other MapSite elements

Structure of Abstract Factory

- Each concrete factory is capable of producing a family of products
- But adding new type of products in each family is hard

Exercise 1

- AbstractFactory
  - Participants:
    - AbstractFactory: AbstractMazeFactory
    - ConcreteFactory: EnchangedMazeFactory, MazeFactory
    - AbstractProduct: AbstractRoom
    - ConcreteProduct: Room, EnchantedRoom, Wall, Door
    - Client: MazeGame
  - Other classes can be added:
    - AbstractWall, AbstractDoor
    - Wall, Door
    - RoomWithOgre, RoomWithTreasure

Exercise 2

- Factory Method
  - Participants:
    - MazeGame
    - ConcreteCreator: BombedMazeGame, EnchangedMazeGame
    - Product: Wall, Room
    - ConcreteProduct: BombedWall, EnchantedRoom
  - Key methods (factory methods):
    - makeMaze()
    - makeWall()
  - Uses an inheritance dimension
  - Interface consistency required between concrete creator and concrete product
  - Can be used for lazy instantiation
  - Provide parallel class hierarchies
  - This pattern is often used by Abstract Factory and Template Methods, but can be used independently too

Abstract Factory vs Factory Method

- Tedious to extending AbstractFactory to produce new product types
- The product/factory type is transparent to the caller and minimizes code maintenance
- Concrete classes are isolated
- Promote consistency among products
- Easy to exchanging product families
- Implicitly uses factory method pattern here
- Concrete factory is usually a Singleton, and may use prototype to create instances
Structure of Singleton

- Defines a class-scoped instance() operation that lets clients access its unique instance
- May be responsible for creating its own unique instance
  - uniqueInstance is a class level static variable
    - you could define more than one instance too, and it is all invisible to the clients/callers
- singletonData is an instance level state variable

Singleton in the Maze

- Singleton: EnchantedMazeFactory, MazeFactory
- Why do we need Singleton?
  - To strictly control the number of instances of a class

Structure of Prototype

- Keeps the standard instances
- Can quickly return new copies of the instances
- Saves effort when large number of similar instances are required

Maze and Prototype

- Participants
  - Prototype: MapSite
  - ConcretePrototype: Door, Room, EnchantedRoom
  - Client: MazePrototypeFactory
- Note that MazeGame is not the client to the prototypes

Structure of Builder

- Does the structure alone provide enough detail of this pattern?

Collaboration of Builder

- Why is the structure alone not enough to describe this pattern?
  - because the the structure of Builder does not depict the behavioral properties.
Maze and Builder

- MazeGenerator
- MazeBuilder
- Maze
- StandardMazeBuilder
- CountingMazeBuilder

Exercise

- What does createMaze() need in order to make use of the following patterns:
  - Factory Method (class scoped)
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton

Putting Them Together

Use the Maze example:
- Can we use Builder and Factory Method together and how?
- What about Abstract Factory and Prototype?
- What about Abstract Factory and Factory Method?
- What about Singleton and Abstract Factory?

Open question:
- What benefits do you get by using these creational patterns?

Tips for Assignment

- Keep in mind that, by employing design patterns, you are looking for a simple solution
- Be concise and precise
- Use point forms whenever possible
- You can use one class diagram to represent a number of patterns if the underlying subdivision is rational
- Do not associate a design pattern with only one class, always describe all the participants in text, the example given on slide 2 would have:
  - Abstract Factory: AbstractMazeFactory
  - Concrete Factory: MazeFactory
  - Concrete Product: Room
  - etc.
- Give a summary (a list) of all identified design patterns, then elaborate in turn along with the appropriate diagrams
- Try to identify non-trivial patterns and try not to reuse the patterns that are implicitly used by other patterns (as seen before)