Notes on Design Patterns

Creational Patterns

- **creational patterns**: abstract the **instantiation process**
  - **class creational pattern**: uses inheritance to vary the class that's instantiated
  - **object creational pattern**: delegates instantiation to another object
- creational patterns gets more important as systems evolve to depend more on **object composition** than **class inheritance**
  - i.e. emphasis shifts away from hardcoding a fixed set of behaviors towards defining a *smaller set of fundamental behaviors* which can be composed into any number of more complex ones
- **Late binding**: when "instantiating an object", don NOT **hard code** the instantiation so that a concrete class will be given as an argument to `new`. This allows flexibility in:
  - **what gets created**:
  - **who creates it**:
  - **how it gets created**:
  - **when it's created**:

BASELINE: Hard-coded

```
Maze *MazeGame::createMaze() {
  // hard-coded constructors
  Maze *maze = new Maze;
  Room *r1 = new Room(1);
  Room *r2 = new Room(2);
  Door *d = new Door(r1, r2);

  maze->addRoom(r1);
  maze->addRoom(r2);
}
```

ABSTRACT FACTORY (C)

- provides an interface for creating families of related or dependent objects without specifying their concrete classes
- pass an "(factory) object" to `CreateMaze`, which can be used to create walls, doors, rooms, etc.

```
class MazeFactory {
  virtual Maze *makeMaze() const {
    return new Maze;
  }
  viirtual Wall *makeWall() const {
    return new Wall;
  }
  ...
};
class EnchantedMazeFactory : public MazeFactory {
  ...
};

Maze *MazeGame::createMaze(MazeFactory& factory) {
  Maze *maze = factory.makeMaze();
  Room *r1 = factory.makeRoom(1);
  Room *r2 = factory.makeRoom(2);
  Door *d = factory.makeDoor(r1, r2);
}
```

FACTORY METHOD (C)

- `createMaze()` calls virtual functions (instead of constructor calls to creaate rooms, walls, etc.)
- create a subclass of `MazeGame` which redefines virtual functions

```
class MazeGame {
  Maze *createMaze();

  // factory methods
  virtual Maze *makeMaze() const { ... }
  virtual Wall *makeWall() const { ... }
  virtual Door *makeDoor() const { ... }
  virtual Room *makeRoom(int i) const { ... }
};

Maze *MazeGame::createMaze() {
  Maze *maze = makeMaze();
  Room *r1 = makeRoom(1);
  Room *r2 = makeRoom(2);
  Door *d = makeDoor(r1, r2);

}

class BombedMazeGame : public MazeGame {
  virtual Maze *makeMaze() const {
    return new BombedWall;
  }
  ...
};
```

BUILDER (C)

- pass an object that can create a new maze **in its entierty** using operations for adding rooms, doors, and walls
- then you can use inheritance to change parts of the maze or the way the maze is built

```
class MazeBuilder {
public:
  virtual void buildMaze();
  virtual void buildRoom(int);
  virtual void buildDoor(int, int);
  virtual Maze *getMaze();
};
Maze *MazeGame::createMaze(MazeBuilder& builder) {
  builder.buildMaze();
  builder.buildRoom(1);
  builder.buildRoom(2);
  builder.buildDoor(1, 2);
  return builder.getMaze();
}

class StandardMazeBuilder : public MazeBuilder {
public:
  virtual void BuildRoom(int n) {
    if (_currentMaze->roomNo(n)) {
      Room *room = new Room(n);
      _currentMaze->addRoom(room);
    }
  }
private:
  Maze *_currentMaze;
};
```

PROTOTYPE (C)

- parameterize `createMaze` by various prototypical room, door, and wall objects, then copy and add them to the maze; then change the maze's composition by replacing these prototypical objects with different ones

```
class MazePrototypeFactory : public MazeFactory {
public:
  MazePrototypeFactory(Maze *, Wall *, Room *, Door *);
  virtual Maze *makeMaze() const;
  virtual Room *makeRoom() const;
  virtual Wall *makeWall() const;
  virtual Door *makeDoor(Room *, Room *) const;
private:
  Maze *_protytypeMaze;
  Room *_protytypeRoom;
  Wall *_protytypeWall;
  Door *_protytypeDoor;
};

MazePrototypeFactory::MazePrototypeFactory(Maze *m,
    Wall *w, Room *r, Door *d) {
  _protytypeMaze = m;
  _protytypeWall = w;
  _protytypeRoom = r;
  _protytypeDoor = d;
}
Door *MazePrototypeFactory::makeDoor(Room *r1,
    Room *r2) const {
  Door *door = _protytypeDoor->clone();
  door->initialize(r1, r2);
  return door;
}
Wall *MazePrototypeFactory::makeWall() const {
  Wall *wall = _protytypeWall->clone();
}

// create prototypyical maze
MazeGame game;
MazeProtytypeFactory simpleMazeFactory(new Maze,
  new Wall, new Room, new Door);
Maze *maze = game.createMaze(simpleMazeFactory);

// to change maze, initialize MazePrototypeFactory
// with different set of prototypes
MazePrototypeFactory bomedMazeFactory(new Maze,
  new BombedWall, new RoomWithBomb, new Door);
```

Structural Patterns

- **COMPOSITE (S)**:
  - most common pattern: trees, ASTs, etc.
- **BRIDGE (S)**:
  - decouple abstraction from its implementation so the two can vary independently

    ```
    class Window {
      void drawText(Text& t) {
        getWindowImpl->drawText();
      }
    protected:
      WindowImpl *getWindowImpl();
    };

    class WindowImpl {
    public:
      void drawText(Text& t) { ... }
    }
    ```
- **ADAPTOR (S)**:
- **FLYWEIGHT (S)**:
  - in case the objects are immutable and there're limited number of possible objects, **share** them