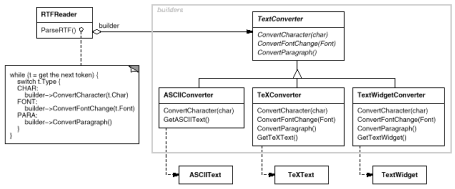
Builder

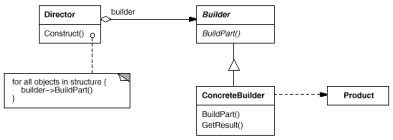
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - e.g., read in Rich Text Format, converting to may different formats on load.



Applicability

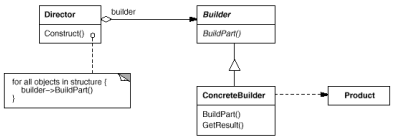
- Use When:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - the construction process must allow different representations for the object that's constructed

Structure



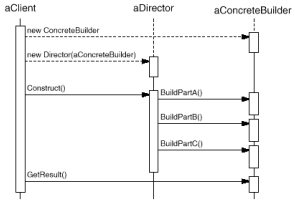
- Builder
  - specifies an abstract interface for creating parts of a Product object
- Concrete Builder
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product

Structure



- Director
  - constructs an object using the Builder interface
- Product
  - represents the complex object under construction.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Collaborations



- The client creates the Director object and configures it with the Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Sample Code

```
public abstract class MazeBuilder {
    public void buildRoom(int r){}
    public void buildDoor(int r1, int direction, int r2){}
    public Maze getMaze(){return null;}
}


public class MazeGame {
    …
    public Maze createMaze(MazeBuilder b) {
        b.buildRoom(1);
        b.buildRoom(2);
        b.buildDoor(1, Direction.North, 2);
        return b.getMaze();
    }
    …
}
```

Sample Code

```
public class StandardMazeBuilder extends MazeBuilder
{
    private Maze currentMaze;

    public Maze getMaze() {
        if( currentMaze==null )
           currentMaze = new Maze();
        return currentMaze;
    }

    …
}
```


Sample Code

```
public class StandardMazeBuilder extends MazeBuilder
{
    …
    public void buildRoom(int r) {
        if( getMaze().getRoom(r) == null ) {
            Room room = new Room(r);
            getMaze().addRoom(room);
            for(int d = Direction.First; d <= Direction.Last; d++)
                room.setSide(d, new Wall());
        }
    }
    …
}
```


Sample Code

```
public class StandardMazeBuilder extends MazeBuilder
{
    …
    public void buildDoor(int r1, int d, int r2) {
        Room room1 = getMaze().getRoom(r1);
        Room room2 = getMaze().getRoom(r2);
        if( room1 == null ) {
            buildRoom(r1);
            room1 = getMaze().getRoom(r1);
        }
        if( room2 == null ) {
            buildRoom(r2);
            room2 = getMaze().getRoom(r2);
        }

        Door door = new Door(room1, room2);

        room1.setSide(d, door);
        room2.setSide(Direction.opposite(d), door);
    }
    …
}
```


Sample Code

```
public class CountingMazeBuilder extends MazeBuilder
{
    private int rooms = 0;
    private int doors = 0;

    public void buildDoor(int r1, int direction, int r2) {
        doors++;
    }

    public void buildRoom(int r) {
        rooms++;
    }

    public int getDoors() { return doors; }
    public int getRooms() { return rooms; }

}
```


Sample Code

```
public class MazeGame
{
    public static void main(String args[]) {
        MazeGame mg = new MazeGame();
        Maze m = mg.createMaze(new StandardMazeBuilder());
        System.out.println(m);

        CountingMazeBuilder cmb = new CountingMazeBuilder();
        mg.createMaze(cmb);
        System.out.println("rooms = "+cmb.getRooms());
        System.out.println("doors = "+cmb.getDoors());
    }

    …
}
```


Sample Code

```
public Maze createMaze(MazeFactory f) {
    Room r1 = f.makeRoom(1);
    Room r2 = f.makeRoom(2);
    Door d  = f.makeDoor(r1,r2);

    r1.setSide(Direction.North, f.makeWall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, f.makeWall());
    r1.setSide(Direction.South, f.makeWall());

    r2.setSide(Direction.North, f.makeWall());
    r2.setSide(Direction.East, f.makeWall());
    r2.setSide(Direction.West, d);
    r2.setSide(Direction.South, f.makeWall());

    Maze m  = f.makeMaze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```

Sample Code

```
public Maze createMaze(MazeBuilder b) {
    b.buildRoom(1);
    b.buildRoom(2);
    b.buildDoor(1, Direction.North, 2);
    return b.getMaze();
}
```


Consequences

- lets you vary a product's internal representation
- isolates code for construction and representation
- gives you control over the construction process

Implementation

- Assembly interface
  - sometimes can just append next element to structure
  - more often must lookup previously constructed elements
    - need an interface for doing this that hides Products
      - cookie of some sort
    - beware order of construction
- Product hierarchy?
  - often no great similarity
  - no great need
  - don't use up a precious inheritance dimension
- abstract v.s. empty methods?
  - empty methods more generally useful
- User-installable product classes

Creational Patterns

- If createMaze() calls virtuals to construct components
  - Factory Method (class scoped)
- If createMaze() is passed a parameter object to create rooms, walls, …
  - Abstract Factory
- If createMaze() is passed a parameter object to create and connect-up mazes
  - Builder
- If createMaze is parameterized with various prototypical rooms, doors, walls, … which it copies and then adds to the maze
  - Prototype
- Need to ensure there is only one maze per game, and everybody can access it, and can extend or replace the maze without touching other code.
  - Singleton