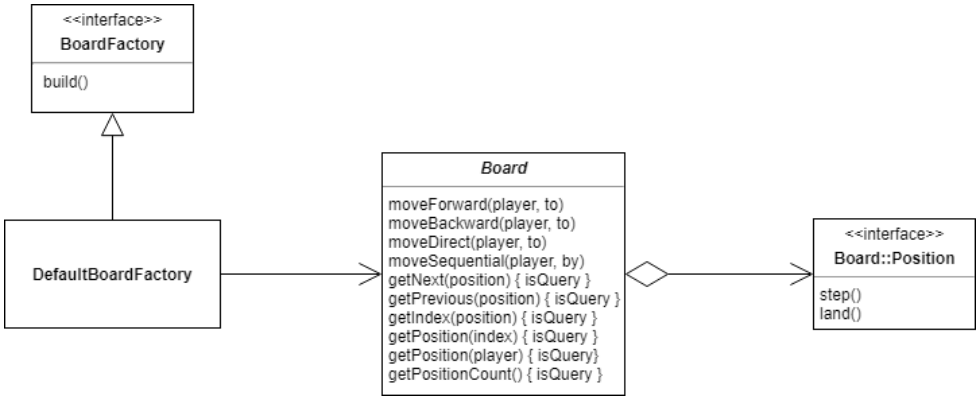
Builder - The Builder pattern is used for the construction of ‘heavy-weight’ instances such as those of the Board and Game classes. The Builder pattern has the following participants:

- **AbstractBuilder** – provides a general interface for constructing a particular type of object.
- **ConcreteBuilder** – provides a specific implementation for constructing the particular type of object.
- **Product** – The particular type of object to be constructed.

I will use the construction of the Board object as a concrete example:



The game class uses a DefaultBoardFactory object to construct a Board object with the correct sequence of positions for standard Property Tycoon board.

The DefaultBoardFactory class can be parameterised with custom properties (including stations and utilities) and card groups. This is so that custom properties and cards can be ‘slotted into’ the standard board in the correct positions.

The real power of using this pattern is the extensibility it provides - a new board layout can be achieved simply by creating a new class that implements BoardFactory, so in theory a board with any number of positions is possible.

Factory Method - The Factory method pattern is used for the construction of many different model class instances. This includes, but is not limited to, Property, Property.Group, Card and Card.Group.

The factory method for each class is usually called create(), or a similar, more meaningful name, if there are multiple factory methods.

Generally within our model, factory methods are used to create and return instances of an abstract class, usually by creating an instance of a package-private subclass. Another common reason to use factory methods is to parameterise other ('third party') objects with a newly created instance. Such factory methods will:

1. Create an instance of the type in question, store it in a local variable.
2. Pass a reference of the newly created object to relevant third party objects.
3. Return the newly created instance.

Passing a reference to the newly created object outside of its constructor avoids the risky action of an object passing a reference to itself whilst it is still being constructed, which could have unintended side-effects.

A concrete example of the factory method in action is the Property.Group.create() method, which:

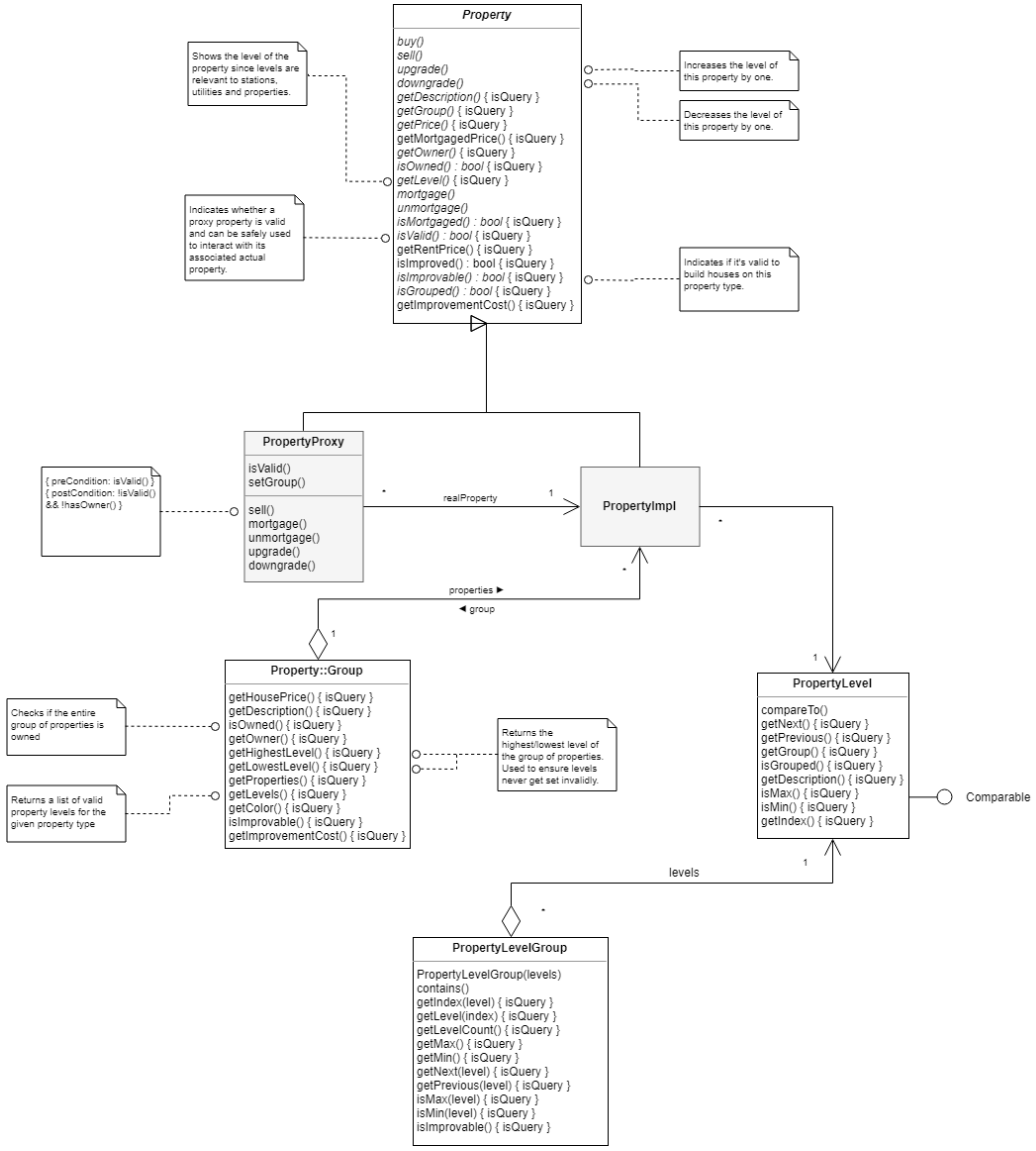
1. Creates an instance of Property.Group.
2. Calls the (protected) setGroup() method on each property passed in as an array parameter.
3. Returns the new group.

Proxy - The proxy pattern is used for cards and properties to provide security against accidental misuse, which is more likely when these objects are shared amongst multiple players throughout the course of a game.
Specifically, these are what are known as 'protection proxies' which provide restricted access to some underlying 'real' object.

The Proxy pattern has the following participants:

- **Subject** – An abstract interface that is used by client code, and implemented by the real subject and proxy classes.
- **RealSubject** – A full implementation of the subject interface, that has no restrictions on interaction (method invocation).
- **Proxy** – A basic implementation of the subject interface the simply delegates to an instance of RealSubject, but may add additional functionality both before and after the delegated method invocation. This can include throwing an error if access to the RealSubject should not be possible.

I will use the Property class as a concrete example:



A given property exists as a single instance of PropertyImpl throughout a games lifetime. This ‘real’ instance is never exposed directly to players. Instead, players interact with them through PropertyPosition and PropertyProxy instances.

All accessor methods are accessible from these proxies at any time, however access to mutator methods is restricted:

- buy() is the only available mutator method for PropertyPosition, which means the only interaction a player has with a property position on the board is to purchase that property. PropertyPosition.buy() returns an instance of PropertyProxy which the player can hold on to.
- PropertyProxy instances provide unrestricted access to their underlying PropertyImpl, until the player chooses to sell the property (by calling sell()). Any attempt to call mutator methods after this results in an error.

The correct behaviour for a Player instance is to discard the PropertyProxy instance after calling sell(), however in the case that a reference is unintentionally maintained, no further interaction with the underlying Property is possible, which means it can be safely reused for the rest of the game.