# Biquardis Design Document

CS 246, Fall 2022

Ed Zhang, Owen Wong, Tom Han

**Overview** **3**

Model 3

View 3

Controller 3

Cohesion and Coupling 4

**Updated UML** **4**

**Design** **4**

Observer Pattern 4

Factory Pattern 5

Model-View-Controller 5

RAII 5

Polymorphism and Inheritance 5

**Resilience to Change** **6**

New Levels (Factory Pattern, Polymorphism) 6

New Blocks (Polymorphism) 6

New Commands and Input Streams (Controller) 6

New Views (Observer Pattern) 6

**Answers to Questions** **6**

**Extra Credit Features** **9**

Smart Pointers 9

Expiring Blocks and Level Five 9

Aliases and macros 9

Game Persistence 9

**Final questions** **9**

# Overview

We used the Model-View-Controller architectural pattern to guide the overall structure of our project. We successfully separated the notions of game logic, data manipulation, and game display, which facilitated our development process and helped decouple our application's individual components.

## Model

**Level:** Our Level implementation followed the Factory design pattern and adhered to the Single Responsibility Principle. All Level type objects were singularly responsible for deciding how to instantiate Block objects.

**Board:** Our Board implementation was primarily responsible for keeping track of various game states. Board owns (see UML Owns-A relationships) an array of unique pointers to Blocks and a Level object, which it used to perform operations on a 2D vector of characters representing the game state (e.g. clearFilledRows method, levelUp and levelDown methods, score-tracking and calculation methods).

**Block:** Our Block implementation consisted of a vector of four, (row, column) Coordinate structures corresponding to the Block's position on a Board, a reference to a Board, and a number of operations that could be performed on the Block (e.g. block rotation using a rotation matrix).

## View

**Observers**: The View component of our implementation of the MVC architectural pattern is singularly responsible for displaying the game. It uses the Observer design pattern. We implemented two separate view classes: one for displaying the game through text (TextView), and one for displaying the game through an XWindow instance (GraphicView). Both view classes inherit from an abstract Observer class, which is referenced by an abstract Subject class. In this case, our Model component plays the role of a Subject with Board being derived from the Subject class, and our Observer view classes observe changes in Board.

## Controller

The Controller component of our implementation was singularly responsible for encapsulating turn-taking logic and game rules, such as alternating between player turns and activating special effects. The controller owns (see UML Owns-A relationship) two Boards, which it uses to keep track of the two players. It uses a selection of private, encapsulated, single-responsibility helper methods to improve code clarity and to facilitate the addition of new gameplay rules (e.g. handleBlockDrop, activateSpecialEffect, etc).

### Cohesion and Coupling

We strived for high cohesion and low coupling in our program. We included no extraneous modules and all our classes serve the unified purpose of implementing the game. We also minimized module interdependence, employing forward declaration and various design patterns that we mention below to limit the information that any module has another. We also separated class responsibilities to minimize the amount of code that needs to be recompiled every time a section of our code is updated.

## Updated UML

1. **MVC:** We learned about MVC after DD1 and decided to implement the architectural pattern as part of our project, which is reflected in our new UML.
2. **Observer Pattern:** Our GraphicView (previously GraphicDisplay) now uses the observer pattern (described in the Design section).
3. **2D Vector of Characters and Coordinate struct:** We made the design choice to not use a Cell class to store character data and coordinate data, since we can access characters in a 2D vector of characters by their row and column anyways. A Block still needs to know its shape and location on a Board though, so we implemented a simple Coordinate struct (see UML) for the Block class.
4. **Non-Virtual Block Rotation:** We devised a way using rotation matrices to rotate blocks without having to hard code rotation in each derived Block class.
5. **Non-Virtual Spawn Block:** We realized that derived level classes only need to set the next block to be spawned, so the logic for spawning a block can be placed in our abstract Level class.

## Design

### Observer Pattern

We implemented the observer pattern on top of the view and model components of our MVC architecture. The Board class acted as the subject, and the TextView and GraphicView classes acted as observers. We used a pull model, so when the Board class notifies its observers to render, the observers retrieve new state data from the Board and compare its own state with the new, pulled state (see UML for more details). If the states differed, then our observers proceeded to render. As such, as little of the screen is redrawn as possible on every single render. Implementing the observer pattern allowed us to decouple display logic from the data-related logic in our model, thus improving the clarity of our code and our resilience to change.

## Factory Pattern

We used the factory pattern to implement level-related logic. We created an abstract Level class with a pure virtual method calculateNextBlock for determining which block should be spawned next, along with what properties (e.g. if the block will disappear in Level 5). Each concrete Level subclass is responsible for implementing calculateNextBlock. When createNextBlock is called, the next block pointer in the level object places itself on its Board. As such, adding new levels is simply a matter of creating a new level class and implementing calculateNextBlock.

## Model-View-Controller

By using the Model-View-Controller architectural pattern, we ensured that the logic corresponding to display, game states, and control flows were encapsulated. This helped us minimize coupling in our codebase and made it easier for us to debug and scale our project as we added new features.

## RAII

We used vectors and smart pointers to bind the management of dynamically allocated resources to components that are cleaned up automatically. For example, a Board binds all heap resources it owns (Blocks, Levels) through unique pointers. Similarly, we implemented the observer pattern through a vector of shared pointers to Observers inside the Board Subjects.

## Polymorphism and Inheritance

Our concrete Level and Block classes are all derived from abstract parents, and as such, we can use virtual methods to execute logic based on a pointer's underlying type at runtime. This significantly reduces the complexity of our codebase, since a single Level or Block pointer can execute the logic of a wide selection of derived classes.

## Resilience to Change

### New Levels (Factory Pattern, Polymorphism)

New levels with new block selection policies can be easily implemented with minimal recompilation by creating a new level subclass and then defining caclulateNextBlock for the new class.

### New Blocks (Polymorphism)

New blocks with new shapes can be easily implemented by creating a new Block subclass.

### New Commands and Input Streams (Controller)

All command processing logic is encapsulated in the controller. When we add new commands, the information is saved in the controller. Additionally, our controller uses a polymorphic istream stack to read input, so we can support arbitrarily many input streams of any type including files and strings.

### New Views (Observer Pattern)

New views can be implemented as a new child of our abstract observer class that implements the pure virtual method, render. Views only need to have a reference to Board in order to pull new state data, so implementing a new view will not increase coupling.

## Answers to Questions

**Question:** *How could you design your system (or modify your existing design) to allow for some generated blocks to disappear from the screen if not cleared before 10 more blocks have fallen? Could the generation of such blocks be easily confined to more advanced levels?*

**Answer:** We set up our Board class to contain a 2D vector of characters, and encapsulate the logic for an individual Biquadris piece in a Block class, which contains a vector of (row, column) Coordinate structures corresponding to cells on the Board following a Has-A relationship. This system enables us to implement the disappearing block logic through an int expiration = 10 field upon a Block's construction. At the beginning of every turn, we iterate through all dropped Block objects and we decrement their expiration counters. Once their expiration counter reaches 0, we remove the block from the board and then we delete the block. To easily confine the generation of such blocks to more advanced levels, we use the Factory design pattern (as described below), which allows each type of level to individually specify how they spawn Blocks. As such, we can spawn Block objects with expiration = 10 to indicate if they are expiring blocks. Then, the block spawning logic of a more advanced level may randomly choose with some probability if a Block is an expiring Block and instantiate it with expiration = 10, but a lower level may only spawn non-expiring Blocks, which by default have their expiration set to -1.

- **Difference from DD1**: Our new answer takes into account that we don't have a Cell class in our new program structure, but otherwise is mostly the same.

**Question:** *How could you design your program to accommodate the possibility of introducing additional levels into the system, with minimum recompilation?*

**Answer:** We employ the Factory design pattern to accommodate the possibility of introducing additional levels into the system with minimal recompilation. We will create an abstract Level class to encapsulate the logic for Block spawning, which can then be implemented independently by its concrete children (e.g., LevelTwo). In doing so, we modularize the unique logic corresponding to the different types of levels in the game, allowing us to avoid cumbersome if-else branches to execute level-related logic. Also, in case some aspect of a Level is bugged, we have fewer areas to debug since Level classes are all self-contained. If we want to introduce additional levels into the system, we can simply create a new class derived from our abstract Level class, and implement the logic of the level in the derived class. This minimizes recompilation. We can skip recompiling the dependents of Level upon changing the implementation of a Level as we only need to recompile the Level itself and link the resulting object code to create a new executable for the program.

**Question:** *How could you design your program to allow multiple effects to be applied simultaneously? What if we invented more kinds of effects? Can you prevent your program from having one else branch for every possible combination?*

**Answer:** We employ the Decorator design pattern to accommodate the simultaneous application of multiple effects. This prevents the need to have one else branch for every possible combination of effects. In this scenario, a Cell class keeping track of each individual square on our game board acts as our ConcreteComponent, and different effects are derived classes of a Decorator abstract class. Both the Decorator and Cell classes inherit from the abstract Component, which contains pure virtual methods to be overridden by Cell and Decorator instances. If we invent more kinds of effects, we must create a new Decorator subclass in order to implement the new effect's logic. We may need to add more pure virtual methods in our base Component class if the new effect affects unanticipated aspects of the

Cell, or we may need to implement the Decorator design pattern elsewhere in our program depending on the nature of the logic

- Example: The blind effect could override char Cell::getVal() method to return a '?' char if the Cell is within a certain range
- Example: The heavy effect affects Block classes, so we may need to implement the Decorator design pattern with respect to blocks as well

**Question:** *How could you design your system to accommodate the addition of new command names, or changes to existing command names, with minimal changes to the source and minimal recompilation? (We acknowledge, of course, that adding a new command probably means adding a new feature, which can mean adding a non-trivial amount of code.) How difficult would it be to adapt your system to support a command whereby a user could rename existing commands (e.g. something like rename counterclockwise cc)? How might you support a "macro" language, which would allow you to give a name to a sequence of commands? Keep in mind the effect that all of these features would have on the available shortcuts for existing command names.*

**Answer:**

To minimize recompilation when adding new commands or changing existing command names, we can create a CommandInterface class that handles the reading of commands, and execute them by interpreting the command strings and calling corresponding methods in the Game class. This way, when we add new commands or rename commands, we only have to recompile CommandInterface without having to recompile Game or any constituents.

In order to design a system that allows users to rename commands, we can use a map<string, string> named aliases. We then implement a command input "rename", which takes in two arguments: oldCommand and alias. After a user calls rename, aliases[alias] will map to oldCommand. Whenever the testing harness receives a command, it first checks if our aliases map contains a key corresponding to the command. If the map does not contain the key, then the command we received is unknown to us and thus we ignore it and ask for a new command.

In order to support a macro language that allows the user to name a sequence of commands, we create a map<string, string> named macros and a command called macro, which takes in arguments until done is read. The first argument is the name of the sequence of commands, and the following arguments are the command names in order. As a side-effect, done is a reserved word and cannot be an alias of a command. After the new macro is read, we map the name of the macro to a string containing space-separated commands in our map, macros. Whenever the testing harness receives a command, it now checks if the command is a macro and if it is not an alias or a default command. If the command is a macro, then we read from an istringstream initialized by the value of the macro in our macros map until the command sequence is finished. To be able to run this sequence of commands, we will wrap the command parsing logic in a function that takes in any istream instead of just reading from std::cin.

## Extra Credit Features

### Smart Pointers

We completed the challenge to complete the entire project, without leaks, and without managing our own memory.

### Expiring Blocks and Level Five

Blocks can randomly be spawned as expiring blocks. After 10 blocks have been dropped, an expiring block will disappear from the board, potentially leaving a hole in a section of blocks that is much harder to clear. Expiring blocks only spawn on level 5, which is a new level with the same features as level 4, except blocks spawn with double the heaviness.

### Aliases and macros

We allow players to define custom aliases that map to existing commands, including their previous aliases. We also support macro languages, where players can give names to a sequence of commands, containing multiplied commands, aliases, and other macros. We ran into difficulties getting macros to support other macros, and we resolved this by representing the input to the program as a stack of istream references. New istreams are pushed onto the stack and popped back off when the Controller finishes reading from them. We stored the macros and aliases in a map of space-delimited commands. Each alias group pushes a new istringstream onto the istream stack, and any non-default command is treated as another macro that is processed in a new istream.

### Game Persistence

At the end of a game, the players are given the option to start a new match. We also track the number of wins each player has throughout the lifetime of the program.

## Final questions

*What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?*

After completing this project and going through the design, execution, testing, and polishing stages, we grew as software engineers and acquired a strong grasp of the mindset and techniques that developers ought to have. Throughout the course of creating the program, we realized the advantages of having at our disposal a multifaceted toolkit of design patterns. We were able to address larger problems with greater efficiency by applying our learnings in software architecture. For example, employing the observer pattern greatly enhanced the scalability and maintainability of our program. It enabled us to systematically tackle the fundamental issue of modeling the execution and responses to object state changes, allowing us to dramatically increase our program efficiency.

In addition to thinking abstractly about how we will engineer our program architecture, we must also consider the implementation and optimization techniques for each granular feature that we create. To successfully create a large, robust, and maintainable program, we found it vital to deliberate in detail about all underlying components that will make up our codebase. We have many tools at our disposal and thus must find the most resilient, most efficient, and most maintainable approach. Ultimately, the project allowed us to tie together and put into practice multiple important aspects of software engineering, including design patterns, planning and execution, task delegation, and both high-level and low-level thinking.

*What would you have done differently if you had the chance to start over?*

After completing the program, we realized the boost to efficiency and robustness that the RAII idiom provides, as well as our program's compatibility with memory management using smart pointers. We agreed that we could have expedited our development process by implementing classes using smart pointers immediately at the start of the project so we can spend less time on explicit memory management.

Additionally, we found it difficult to recall some of the implementation details concerning private and protected methods we discussed while writing our UML for DD1, since the UML diagram only required us to record public methods for each class. Given a chance to start over, we agreed we would have spent more time making a full UML diagram containing private and protected fields and methods before DD1 for our own sanity when working on DD2.