GameDroid: A GameBoy Emulator for Android

COMP 3004 – Deliverable 3

Team CreativeName:

Matthew Penny (100937331)
Hammad Asad (100854593)
Alan Wu (100942033)
Brendan Marko (100729104)

1.0 – Project architecture

Our project is an emulator for the Nintendo GameBoy (a handheld videogame system first released in 1989). For an emulator to work, it must simulate the hardware present in the original system. Due to the complexity of a project such as this, we employed an object-oriented architecture for the sake of encapsulation and maintainability. This abstracts away implementation details using public interfaces. Our emulator is composed of two distinct parts: the core emulator itself and the user interface. These two parts are entirely self-contained with the exception of some connectors (public functions) acting as bridges between the two (i.e., for sending user input to the emulator core and rendering the GameBoy display in the user interface). Each emulated piece of hardware has a corresponding component (object) which is responsible for updating its own state. For example, the emulated CPU executes game code but its only public function simply runs the next instruction, abstracting away how it is done. Each emulated piece of hardware implements the same common interface, thus the emulator core is able to interact with all them in the same way. This decision makes it possible to modify the implementation of any component without affecting other, unrelated ones (assuming the public interface does not change). It also allows the addition of new components in the future with minimal changes to the existing codebase (e.g., if we ever decided to expand the emulator to include support for sound).

Some of the components of the application must communicate with each other (e.g., the emulated CPU and LCD controller). In these situations, the mediator pattern is used. Inter-component communication is carried out through the MMU component. This pattern reduces dependencies and coupling between components (facilitating easier code maintenance) and also implements the functionality of the GameBoy's memory-mapped I/O (a functional requirement for the project).

The most critical part of any emulator is the CPU since it is the component responsible for executing code. The emulator's CPU is an interpreter that maps assembly instructions to functions which perform equivalent actions to modify the state of the emulated GameBoy. The interpreter steps through GameBoy games' code one instruction at a time, executing the corresponding function for each one. This is important because the code the emulator runs is dynamic and changes at run-time (different game ROM files may be loaded). The interpreter style models how the processor in a real GameBoy actually behaves. Modelling the hardware as closely as possible results in more accurate emulation and makes it easier to follow available documentation. It also improves code readbility, since those familiar with assembly and the internal operation of processors will be accustomed to the control flow. Although the interpreter style can have low performance in some cases, the age and simplicity of the hardware being emulated allows us to achieve good performance regardless.

Due to technical limitations of the time, the GameBoy is only able to address 32KB of cartridge data at a time. As a solution to this, many game cartridges contain extra hardware to switch different banks of game data in and out of this 32KB “window”. These extra pieces of hardware are called memory bank controllers (MBCs) and it is important to emulate them, otherwise our application will only be capable of playing the most limited of GameBoy games. MBCs are implemented using the template pattern. Common functionality is implemented in the abstract base component, and functionality specific to the individual MBC types is implemented in their respective components. The base/template component uses this MBC-specific functionality when providing game data to the emulator. The template pattern reduces code duplication and makes it simple to maintain the implemented MBCs or add support for other ones.

Each piece of hardware runs in parallel in a real GameBoy. However, emulating using this approach and accurately keeping every component accurately synchronized is difficult. For this reason, the emulator core is single-threaded and updates one component at a time using a batch sequential style. First, the emulated CPU interprets one instruction and then the internal timer and LCD controller are simulated, one after the other, for the amount of time it would have taken to execute the instruction on real GameBoy hardware. This is simpler to implement and maintain since some components rely on the output of calculations performed by others. By updating one component at a time until their states “catch up” to each other, it can always be assured that a given component will have the data it needs to properly carry out its function.

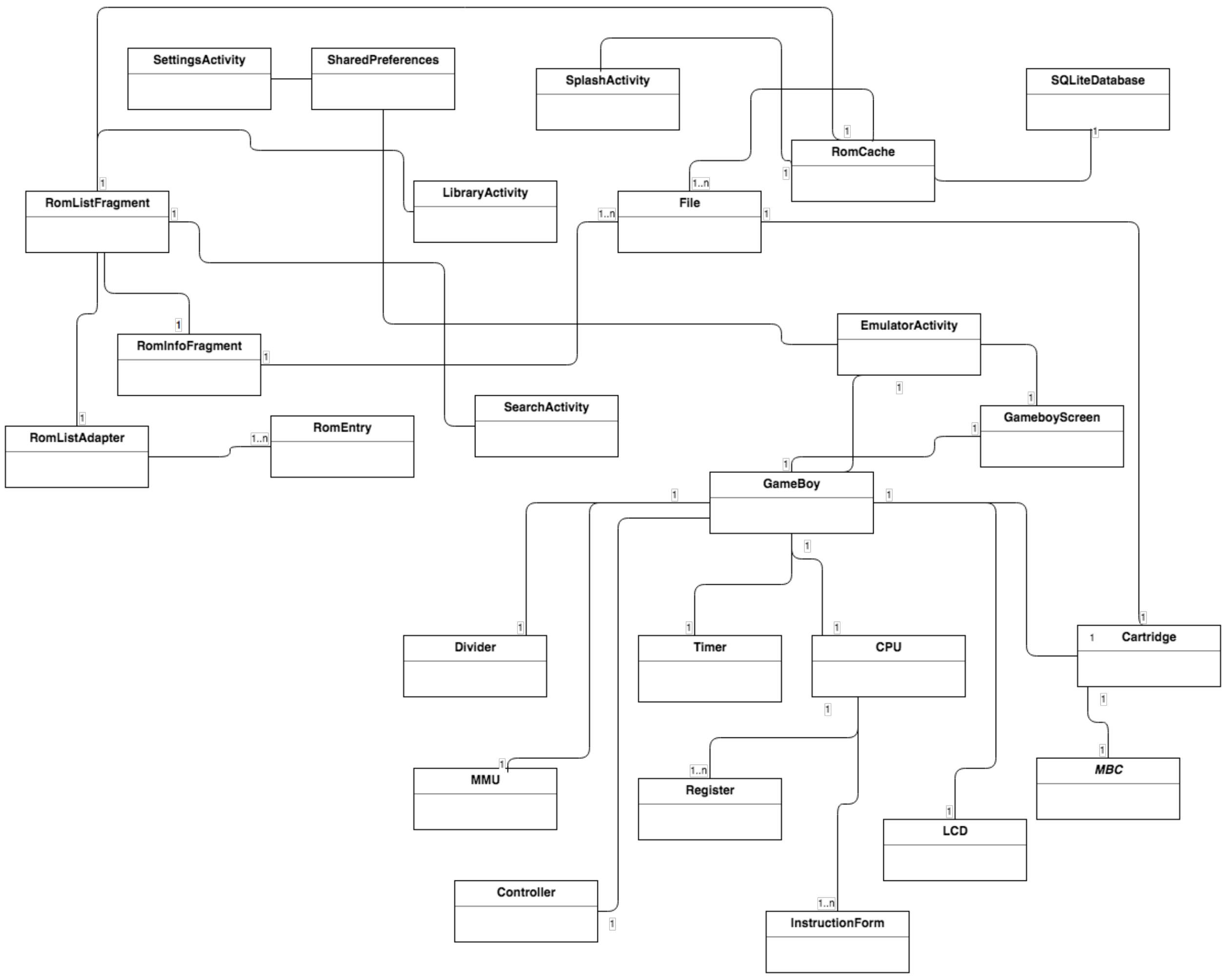
The timing of some behavior of the emulator is unpredictable which makes a batch sequential update loop ill-suited. An event-based architecture is used for these cases. Rather than continuously polling the internal state, components which act on unpredictable events are instead notified when they occur. For example, whenever the user presses a button in the user interface (e.g., to interact with a running game), a function is called to handle the event (e.g., update the state of the emulated game controller). This style is also employed for rendering GameBoy output. The emulated GameBoy notifies the user interface when there is a frame that needs to be drawn, and the UI does not waste time constantly checking if there is new output. An observer pattern like this is also used for updating the various game library tabs. Users are able to change their contents by marking a game as a favorite or deleting it. When this happens, the container which holds the game ROM lists is notified of the change and subsequently refreshes all of the tabs (reflecting whatever change was made). An event-based architecture affords improvements by only carrying out actions when they need to be done, and is therefore very beneficial for a performance-critical application such as an emulator. The observer pattern also reduces coupling and promotes component reuse (the same component is used to display ROM files in the library view and search view).

Application usability played a key role in the development of the architecture for this project. The user interface presents lists of GameBoy game ROM metadata to users. This metadata is read from the ROM files themselves, which are kilobytes in size. Reading many small files is very slow, so the information is cached in a database (which noticeably improves application startup times when large amounts of games are in the library). Access to the database is performed using the façade pattern so that individual components do not need to perform the database queries themselves (or know which queries are being made at all). It makes it much simpler to retrieve information and allows the metadata storage method to be changed later if the need arises.

Multiple components make use of ROM metadata (the lists in the game library, ROM search view, and ROM info dialog all display it) and it is important that the data remains consistent between them so that the application does not display old information. We use the singleton pattern to accomplish this. The ROM metadata list is stored in a component which can only be instantiated once. This way, the list is only ever updated and accessed using the singleton and components always reflect the latest information.

GameDroid’s architecture is motivated by a desire to ease code maintenance, improve readability, reduce coupling, and allow for extensibility. Several architectural styles and design patterns are employed, which come together to achieve this goal and support the project’s requirements.

1.1 – Architecture diagram



2.0 – Project design

GameDroid is designed to be easily understandable, maintainable, and extensible. Its design patterns, component structures, and loose coupling facilitate this. Additinally, all classes (as well as the Android APIs used) reside on the client, ensuring snappy performance.

CPU emulation is a crucial component of the emulator. The only way to control the GameBoy is through CPU instructions, and therefore no game would run without proper CPU emulation. As mentioned in part 1, the emulated CPU is implemented using the interpreter architecture. The CPU keeps track of a program counter, which is an offset from the beginning of the ROM that points to the instruction that is currently being executed. Each instruction has a unique operation code (opcode), which is a 1-byte integer stored in the ROM data of the game cartridge. Different instructions require a different number of operands, which are put immediately after the instruction in the ROM. During emulation, the CPU inspects the opcode located at the current program counter position, deduces the opcode’s corresponding instruction, loads the appropriate operands for the instruction, carries out the instruction, and finally increments the program counter so it points to the opcode of the next instruction.

There are a total of 501 instructions that the GameBoy’s CPU supports. Many of the instructions ultimately carry out the same operation, with the only difference being the way they interpret their operands. For example, there is an instruction that loads its operand directly into register A, and there is another instruction which uses its operand to construct a pointer and then loads the value pointed to by said pointer into register A. While some implementations in other emulators simply have 501 distinct methods (one for each opcode) GameDroid employs a few mechanisms to only require one method for each group of similar instructions. Each instruction group has a corresponding class which implements the InstructionRoot interface. These implementations operate on Cursors, which provide two operations: read and write. Cursors abstract away the actual location that they read from or write to, allowing InstructionRoots to achieve their generality. Going back to the “load” example mentioned above, the root implementation would be “read from the second operand and write the resulting value into the first operand”. To implement the two different instructions, one would simply pass two different sets of cursors to the same root. The cursor for the second case would take care of the pointer construction and dereferencing, and the cursor for the first case would return the operand in its raw form. Using a lookup table, the CPU class is efficiently able to find the correct instruction root for a given opcode and the appropriate way of constructing the set of cursors to pass to its root for execution. This design eliminates code duplication, helps ensure consistency within instruction groups, and makes adding new instructions easy.

The MMU is another important part of the emulator. As mentioned in part 1, it uses the mediator pattern. Every piece of hardware connected to the GameBoy is interfaced with via memory accesses, and the MMU acts as a bridge to facilitate all the functionalities that game developers use. While some memory addresses simply map to RAM, many addresses have special behaviors. Writing to one of these special addresses might indicate that the emulator needs to enable or disable some device, or invoke some functionality of the device. In a way, read and write to special addresses are analogous to API calls exposed to game developers. The outcome of reading to and writing from different memory locations drastically differs, thus the actual implementations of the behaviors resides in different classes. Each of these classes implements the MemoryMappable interface, enabling the MMU to communicate with them each in the same way. The MMU simply dispatches requests for reads and writes to the different device based on the target address. The device classes themselves are responsible for interpreting the meaning of a read/write address and any values written. This design allows a clean separation between devices, and keeps the responsibility of the MMU simple. While the MMU could theoretically handle all of the address translation down to the device level, that design would cause device classes to expose more public methods and make their implementations less localized, reducing encapsulation.

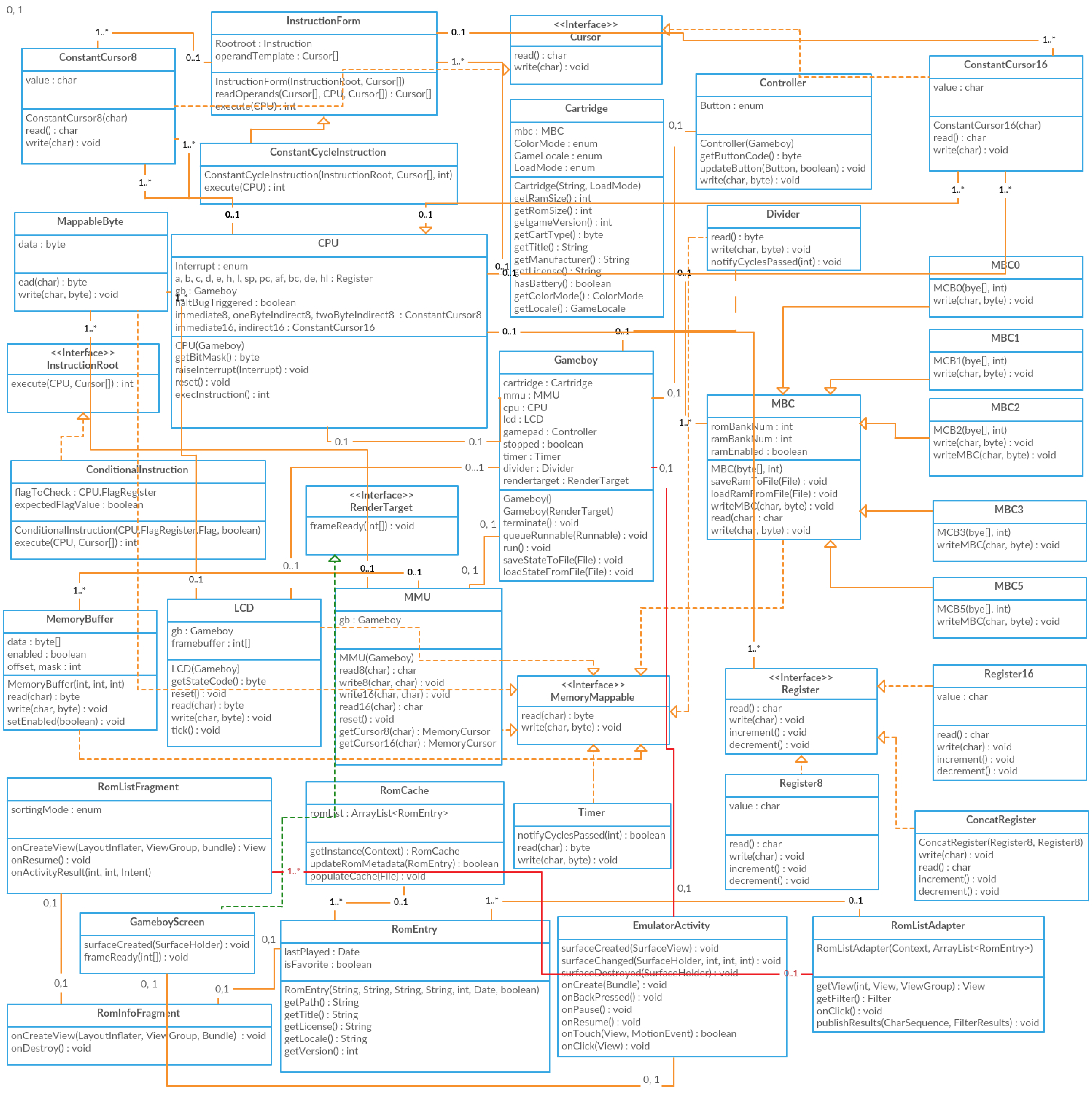
The emulator is designed to run in a dedicated thread and is controlled during execution via inter-thread communication. When a frame is ready to be displayed, the emulator uses the observer pattern to notify the Android portion of the code, and the frame is drawn on the phone’s screen. Similarly, when the user presses buttons on the Android UI, the emulator is notified about these presses. Using the observer pattern to keep a clean separation between the Android portion of the project and the emulator has helped with implementation and testing. Work on both portions of the project can happen concurrently without conflict, and emulator tests do not have to interact with any portion of the Android code. This separation also makes the emulation code completely portable. If the need arises, the emulator core can be easily reused on a different Java-based platform (i.e., as a desktop application).

One plausible way in which the system can evolve in the future is by modifying the emulator to support GameBoy Color (GBC) games. Adding this extra functionality should be straightforward thanks to the clear separation between internal devices. The major difference between the GameBoy and the GameBoy Color is that the latter handles graphical data differently and allows games to change the clock speed of the CPU. Supporting the GBC would involve the implementation of additional registers in different device classes. The design of the MMU makes adding new devices (and new functionalities to existing devices) simple. All that needs to change is the dispatch portion of the MMU (to be able to handle reads/writes) and the implementation of the devices themselves. The logic for video rendering would also have to change to support the GBC. However, because all the logic for rendering is encapsulated inside a single device class (the LCD class), the changes for it would be localized, requiring little to no modification of the rest of the code.

Another possible modification to the system is the addition of support for the GameBoy's lesser used, non-core expansion hardware. This includes link cable communication and external devices specific to special, obscure cartridges. Similar to supporting GameBoy Color games, this would also be hassle-free. GameDroid's loosely coupled design minimizes the overhead and work required for change.

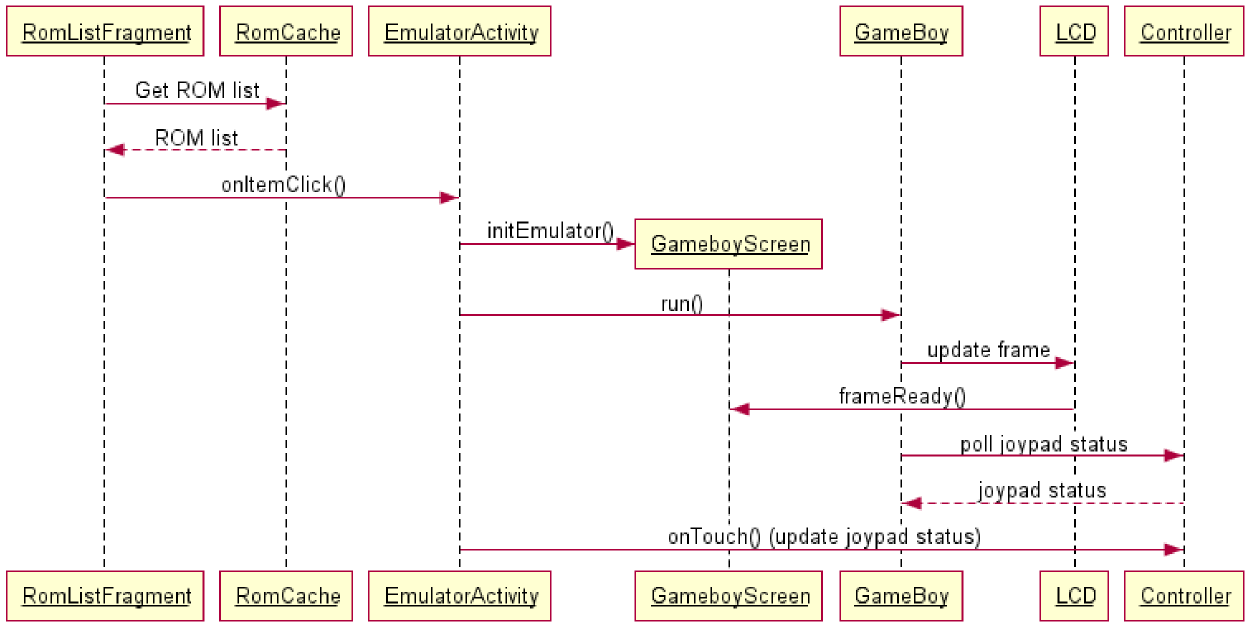
2.1 – Class diagram

See next page.

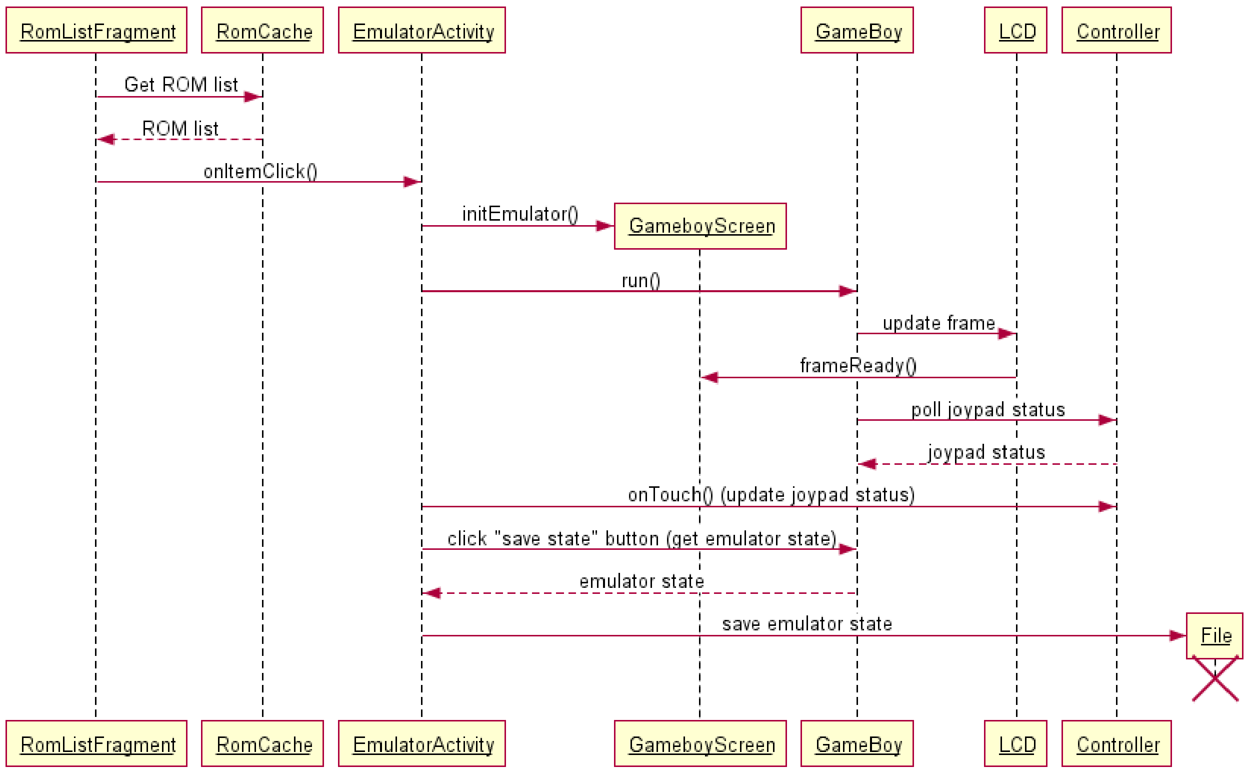

0, 1
InstructionForm
Rootroot : Instruction
operandTemplate : Cursor[]
InstructionForm(InstructionRoot, Cursor[])
readOperands(Cursor[], CPU, Cursor[]) : Cursor[]
execute(CPU) : int
<<Interface>>
Cursor
read() : char
write(char) : void
ConstantCursor8
value : char
ConstantCursor8(char)
read() : char
write(char) : void
ConstantCycleInstruction
ConstantCycleInstruction(InstructionRoot, Cursor[], int)
execute(CPU) : int
Cartridge
mbc : MBC
ColorMode : enum
GameLocale : enum
LoadMode : enum
Cartridge(String, LoadMode)
getRamSize() : int
getRomSize() : int
getgameVersion() : int
getCartType() : byte
getTitle() : String
getManufacturer() : String
getLicense() : String
hasBattery() : boolean
getColorMode() : ColorMode
getLocale() : GameLocale
Controller
Button : enum
Controller(Gameboy)
getButtonCode() : byte
updateButton(Button, boolean) : void
write(char, byte) : void
ConstantCursor16
value : char
ConstantCursor16(char)
read() : char
write(char) : void
MappableByte
data : byte
ead(char) : byte
write(char, byte) : void
CPU
Interrupt : enum
a, b, c, d, e, h, l, sp, pc, af, bc, de, hl : Register
gb : Gameboy
haltBugTriggered : boolean
immediate8, oneByteIndirect8, twoByteIndirect8 : ConstantCursor8
immediate16, indirect16 : ConstantCursor16
CPU(Gameboy)
getBitMask() : byte
raiseInterrupt(Interrupt) : void
reset() : void
execInstruction() : int
Divider
read() : byte
write(char, byte) : void
notifyCyclesPassed(int) : void
MBC0
MCB0(bye[], int)
write(char, byte) : void
MBC1
MCB1(bye[], int)
write(char, byte) : void
MBC2
MCB2(bye[], int)
write(char, byte) : void
writeMBC(char, byte) : void
MBC3
MCB3(bye[], int)
writeMBC(char, byte) : void
MBC5
MCB5(bye[], int)
writeMBC(char, byte) : void
<<Interface>>
InstructionRoot
execute(CPU, Cursor[]) : int
Gameboy
cartridge : Cartridge
mmu : MMU
cpu : CPU
lcd : LCD
gamepad : Controller
stopped : boolean
timer : Timer
divider : Divider
rendertarget : RenderTarget
Gameboy()
Gameboy(RenderTarget)
terminate() : void
queueRunnable(Runnable) : void
run() : void
saveStateToFile(File) : void
loadStateFromFile(File) : void
MBC
romBankNum : int
ramBankNum : int
ramEnabled : boolean
MBC(byte[], int)
saveRamToFile(File) : void
loadRamFromFile(File) : void
writeMBC(char, byte) : void
read(char) : char
write(char, byte) : void
ConditionalInstruction
flagToCheck : CPU.FlagRegister
expectedFlagValue : boolean
ConditionalInstruction(CPU.FlagRegister.Flag, boolean)
execute(CPU, Cursor[]) : int
<<Interface>>
RenderTarget
frameReady(int[]) : void
MemoryBuffer
data : byte[]
enabled : boolean
offset, mask : int
MemoryBuffer(int, int, int)
read(char) : byte
write(char, byte) : void
setEnabled(boolean) : void
LCD
gb : Gameboy
framebuffer : int[]
LCD(Gameboy)
getStateCode() : byte
reset() : void
read(char) : byte
write(char, byte) : void
tick() : void
MMU
gb : Gameboy
MMU(Gameboy)
read8(char) : char
write8(char, char) : void
write16(char, char) : void
read16(char) : char
reset() : void
getCursor8(char) : MemoryCursor
getCursor16(char) : MemoryCursor
<<Interface>>
MemoryMappable
read(char) : byte
write(char, byte) : void
<<Interface>>
Register
read() : char
write(char) : void
increment() : void
decrement() : void
Register16
value : char
read() : char
write(char) : void
increment() : void
decrement() : void
Register8
value : char
read() : char
write(char) : void
increment() : void
decrement() : void
ConcatRegister
ConcatRegister(Register8, Register8)
write(char) : void
read() : char
increment() : void
decrement() : void
RomListFragment
sortingMode : enum
onCreateView(LayoutInflater, ViewGroup, bundle) : View
onResume() : void
onActivityResult(int, int, Intent)
RomCache
romList : ArrayList<RomEntry>
getInstance(Context) : RomCache
updateRomMetadata(RomEntry) : boolean
populateCache(File) : void
Timer
notifyCyclesPassed(int) : boolean
read(char) : byte
write(char, byte) : void
GameboyScreen
surfaceCreated(SurfaceHolder) : void
frameReady(int[]) : void
RomEntry
lastPlayed : Date
isFavorite : boolean
RomEntry(String, String, String, String, int, Date, boolean)
getPath() : String
getTitle() : String
getLicense() : String
getLocale() : String
getVersion() : int
EmulatorActivity
surfaceCreated(SurfaceView) : void
surfaceChanged(SurfaceHolder, int, int, int) : void
surfaceDestroyed(SurfaceHolder) : void
onCreate(Bundle) : void
onBackPressed() : void
onPause() : void
onResume() : void
onTouch(View, MotionEvent) : boolean
onClick(View) : void
RomListAdapter
RomListAdapter(Context, ArrayList<RomEntry>)
getView(int, View, ViewGroup) : View
getFilter() : Filter
onClick() : void
publishResults(CharSequence, FilterResults) : void
RomInfoFragment
onCreateView(LayoutInflater, ViewGroup, Bundle) : void
onDestroy() : void
1..*
0..1
0,1
0, 1
0...1
0..1
0,1

2.2 – User scenario sequence diagrams

Loading a game ROM file



Saving a game state



3.0 – Distribution of work

Matthew Penny:

<u>Emulator core (creativename.gamedroid.core):</u>

- Parsing game ROM files (allocating memory for games, and reading metadata and executable code)
    - Cartridge class
- Architecture and implementation of cartridge hardware emulation for RAM and ROM bank switching
    - Abstract MBC class and its concrete subclasses (MBC0, MBC1, MBC2, MBC3, and MBC5)
- Gamepad emulation (allows emulated games to read user input)
    - Controller class
- CPU instruction implementation
    - LDI, LDD, NOP, SCF, CCF, CPL, CP, INC8, INC16, DEC8, DEC16, ADD16, ADDSP, AND, XOR, OR, PUSH, POP, SWAP, BIT, RES, SET, JP, JPNZ, JPZ, JPNC, JPC, JR, JRNZ, JRZ, JRNC, JRZ, CALL, CALLNZ, CALLZ, CALLNC, CALLC, RST, RET, RETNZ, RETZ, RETNC, RETC, RETI, HALT, STOP, EI, DI, and DAA static inner classes of the CPU class
- Architecture for interrupt handling
    - Implemented in the CPU and MMU classes
    - Also implemented the vertical blank interrupt (in the LCD class), LCD status interrupt (in the LCD class), and joypad interrupt (in the Controller class)
- GameBoy graphics architecture, state machine, rendering, and timing
    - LCD class
- SRAM game saves (those natively supported by GameBoy games)
    - Implemented in the MBC class
- Save states (“save anywhere” feature)
    - Implemented in the GameBoy class
- Stack operations
    - Implemented in the CPU and MMU classes

<u>User interface (creativename.gamedroid.ui):</u>

- Game library UI (tabbed view, ROM sorting, and favoriting)
    - LibraryActivity, RomListFragment, and RomListAdapter classes
- Caching layer for saving large amounts of ROM metadata on startup
    - SplashActivity and RomCache classes
- Connecting UI buttons to emulator core (gamepad and save/load state buttons)
    - EmulatorActivity class

Alan Wu:

<u>Emulator core (creativename.gamedroid.core):</u>

- GameBoy memory mapping architecture and implementation
    - Cursor, ConstantCursor8, ConstantCursor16, and MMU classes

- CPU architecture and implementation
    - Register, Register8, Register16, and ConcatRegister, and CPU classes
- CPU instruction architecture and implementation (including operand translation and timing)
    - ConstantCycleInstruction, InstructionRoot, and InstructionForm classes
    - LD, RL, RLC, RR, RRC, SLA, SRA, and SRL static inner classes of the CPU class
- CPU timer interrupt implementation
    - Timer and Divider classes
- Architecture for rendering to the UI
    - RenderTarget interface
- Rewind feature
    - GameBoy and LCD classes
- CPU emulation accuracy (obscure bugs)
    - Reading and watching the execution of GameBoy assembly

<u>User interface (creativename.gamedroid.ui):</u>

- Displaying rendered GameBoy display in the UI and managing the emulation thread
    - EmulatorActivity, GameboyScreen, and EmulatorFragment classes

Hammad Asad:

<u>Emulator core (creativename.gamedroid.core):</u>

- CPU instruction implementation
    - ADD8, ADC, SUB, and SBC static inner classes of the CPU class

<u>User interface (creativename.gamedroid.ui):</u>

- Information dialog for viewing ROM information and managing associated files
    - RomInfoFragment class

Brendan Marko:

<u>User interface (creativename.gamedroid.ui):</u>

- ROM file metadata display
    - RomListAdapter and RomEntry classes
- ROM file searching
    - SearchActivity class
- Gamepad layout
    - EmulatorActivity class
- User-configurable settings
    - SettingsActivity and EmulatorActivity classes
- Obtaining proper permissions
    - SplashActivity class
- Application icon and splash screen logo