DESIGN

[L7P1]

The View from the Top: Architecture

A software project is initially concerned with *what* to build (i.e establishing requirements and deciding system features). This is called the *analysis* phase.

At some point, attention shifts to figuring out *how* to build it (i.e. internal details of the implementation). This is called the *design* phase.

There is no clear line to mark the end of one phase and the start of the other. Often, a decision taken during the analysis phase of one project could also be taken during the design phase of another project. For example, the decision on the choice of the development platform for system implementation can be taken during the analysis phase of one project and during the early design phase of another project.

With simple projects, we can 'design using code': By using a mental image of how the program ought to be structured, we can start coding without having to create a design. In fact, code represents a very detailed level of design; it is after all the 'blueprint' that will be compiled into the executable product. It would appear that coding is the most efficient way to design without the need to draw numerous diagrams. However, this approach does not scale up to larger, more complex, multi-person projects.

The design phase is generally divided into two levels:

- *High-level* design: In high-level design, designers take decisions considering the overall system, i.e. decisions made here affect the system as a whole. The high-level overall description of a system includes the top-level structure of subsystems, as well as the roles of these subsystems and the interactions between them.
- *Detailed* (also called *low-level*) design: In detailed design, designers work at the module or sub-system levels.

Software architecture

The software architecture shows the overall organization of the system. It can be viewed as a very high-level design. It should be a simple and technically viable structure that is well-understood and agreed-upon by everyone in the development team, and it forms the basis for the implementation.

Here's a formal description:

> *As the size and complexity of software systems increases, the design problem goes beyond the algorithms and data structures of the computation. Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives. This is the software architecture level of design.*
> *[An Introduction to Software Architecture, David Garlan and Mary Shaw]*

The architecture is typically designed by the *software architect*, who provides the technical vision of the system and makes high-level (i.e. architecture-level) technical decisions on the project.

System architecture addresses the big picture and usually consists of a set of interacting components that fit together to achieve the required functionality. Figure 1 gives a possible architecture for the *Minesweeper* game (screenshot given below). The high-level components are:

GUI: Graphical user interface

TextUi: Textual user interface

ATD: An automated test driver used for testing the game logic

MSLogic: computation and logic of the game

MSStore: storage and retrieval of game data (high scores etc.)

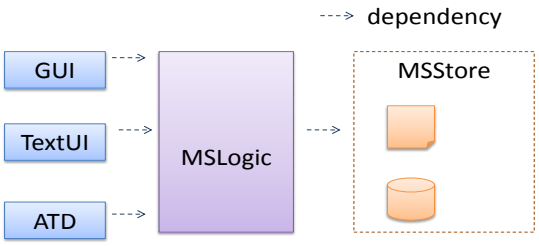


Figure 1. A possible architecture for Minesweeper

Architecture diagrams

Architecture diagrams, such as the one given in Figure 1, are free-form diagrams. There is no universally adopted standard notation for architecture diagrams. Any symbol that reasonably describes the architecture may be used. However, the indiscriminate use of double-headed arrows (<----->) to show interactions between components is discouraged. If double headed arrows were used in Figure 1, it would not be possible to describe the dependency among the components. Furthermore, the representation of arrows should be specified. Do they indicate a dependency, data flow, or something else? Use arrows of different styles (e.g. dashed/solid, thin/thick) to indicate the different representations.

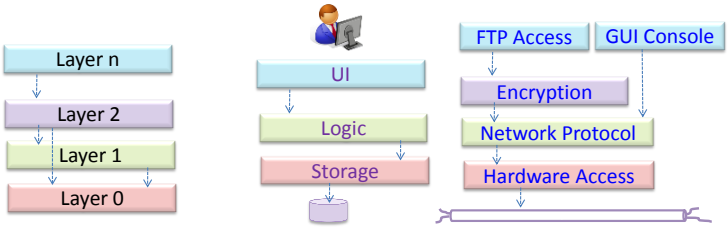
Architectural styles

Given next are some examples of common architectural styles.

The n-tier architectural style

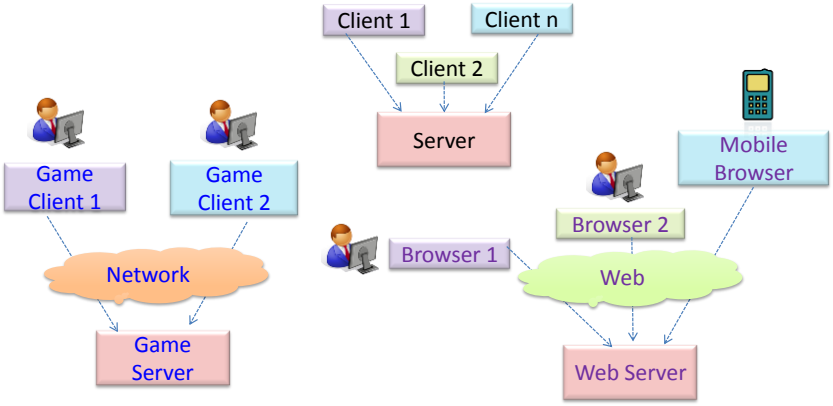
The main characteristic of an n-tier architectural style is that higher layers make use of services provided by lower layers. Lower layers are independent of higher layers.

Operating systems and network communication software often use this style.



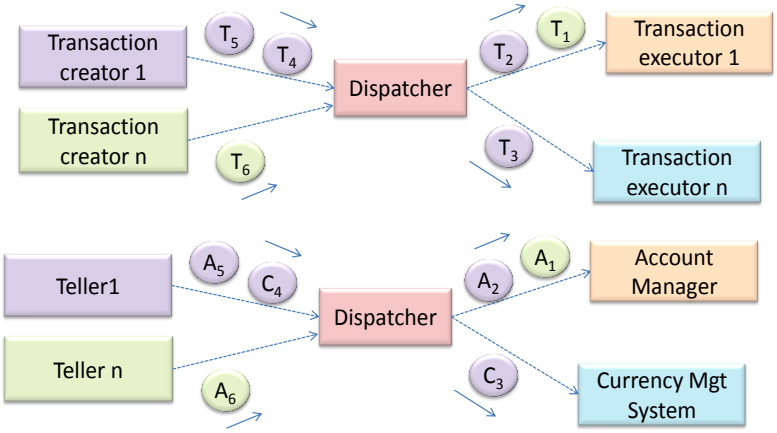
The client-server architectural style

This is an architectural style for distributed applications. The main characteristic of the client-server style is the presence of at least one component playing the role of a server, with at least one client component accessing the services of the server.



The transaction processing architectural style

The main characteristic of a transaction processing architecture style is that the workload of the system is broken down to a number of *transactions*. These transactions are then given to a "dispatcher" that controls the execution of each transaction. Task queuing, ordering, "undo" etc. are handled by the dispatcher.

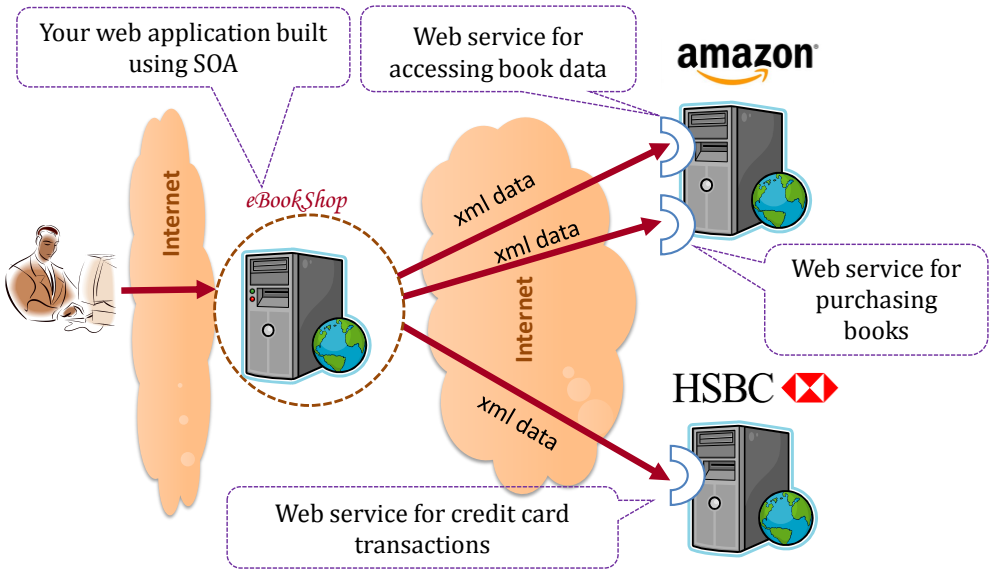


The Service-oriented architectural style

The *service-oriented architecture* (SOA) is a relatively recent architectural style for distributed applications. An SOA is essentially an application built by combining functionalities packaged as programmatically accessible services. A prerequisite for SOA is the interoperability between distributed services, which may not even be implemented using the same programming language. A common way to implement SOAs is through the use of *XML web services* where the

web is used as the medium for the services to interact, and XML is used as the language of communication between service providers and service users.

As an example, suppose that Amazon.com provides a web service for customers to browse and buy merchandize, while HSBC provides a web service for merchants to charge HSBC credit cards. Using these web services, an 'eBookShop' web application can be developed that allows HSBC customers to buy merchandize from Amazon and pay for them using HSBC credit cards. Since both Amazon and HSBC services follow the SOA architecture, their web services can be reused by the web application, even if all three systems use different programming platforms.



Event-driven architectural style

An event is a notable occurrence that happens inside or outside the software, such as the user clicking a button, a timer running out, minimizing a window, etc. In the event-driven architectural style, events are detected and communicated to the affected components. These components then react to the events, if required. For example, when the 'button clicked' event occurs, functions that are tied to the button-click event is activated. This architectural style is often used in GUIs.

Other architectural styles

Other well-known architectural styles include the *pipes-and-filters* architectures, the *broker* architectures, the *peer-to-peer* architectures, and the *message-oriented* architectures.

Most applications use a mix of these architectural styles or adopt a style that is customized to the requirements. For example, an application can use a client-server architecture where the server component comprises several layers, i.e. it uses the multi-layer architecture.

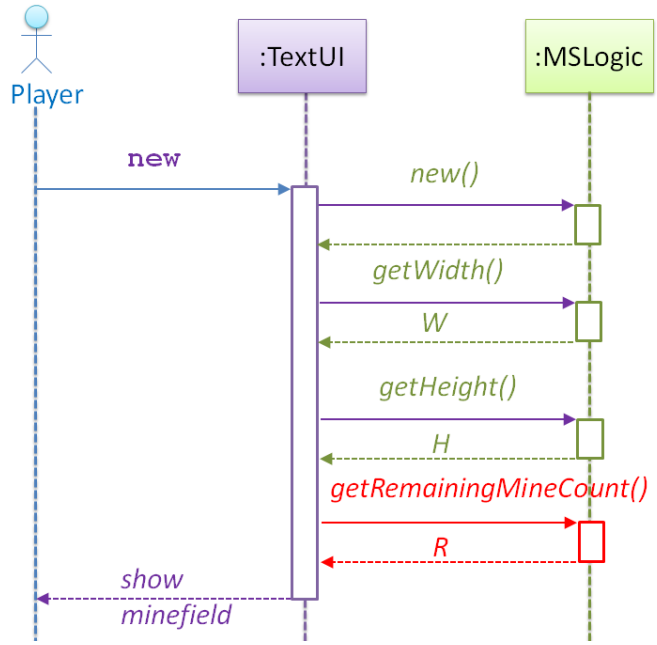
APIs

The Minesweeper architecture given in Figure 1 depicts the high-level components of the system and the anticipated dependencies between them. However, the architecture does not describe how components communicate with each other, i.e. the operations that each component should provide so that they can work with each other to achieve the features of the system. In other words, the exact *interface* for these components needs to be defined. Here, the term 'interface' means the list of public operations supported by a component and what each

operation does. Once the interface is decided, implementation of various components can be done simultaneously. The interface of a component is also called the *Application Programming Interface (API)*.

An API is a contract between the component and its users. It should be well-designed (i.e. should cater for the needs of its users) and well-documented. As an example, refer to the API of the Java String component given here
http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

We can use UML *sequence diagrams* to analyze the required interactions between components. Given below is an example.



As we analyze the interactions between components using sequence diagrams, we discover the API of those components. For example, the diagram above tells us that the MSLogic component API should have the methods:

new() getWidth:int getHeight():int getRemainingMineCount():int

Different approaches to design

Top down vs bottom up

The approach taken in this handout is to start from the very high-level by viewing the system as one big black box and then break it into a handful of smaller components. Each of these components can be broken down further into a small number of sub components, and the process continues until the complete detailed design is obtained. This approach is called *top-down design*. When using top down design, low level details are not considered until much later in the design process. Moreover, the low level details of a given component can be worked out by those working on that component, without getting the whole team involved. The top-down approach is often used when creating a big product from scratch.

The reverse of the top-down approach is *bottom-up*. That is, starting with lower level details (e.g. data structures, storage formats, functions etc.) and progressively grouping them together to create bigger components. This approach is often adopted when the system is small or when

developing a variant of a previous product whereby a large collection of reusable assets can be used in the new product.

Agile vs full-design-up-front

How much design is deemed sufficient before the coding phase can start? In the industry, there are two schools of thought:

- *Agile-design* camp considers it enough if the design can support the feature that is going to be implemented in the immediate future. They argue that since a product's feature set can evolve during its lifetime, there is no use trying to cater for all of them from the very beginning.
- *Full-design-up-front* camp considers a full design so as to support the entire feature set and even possible future features before we start implementing it.

As with the case with most alternative approaches, there are pros and cons for each and a mixture is often the best.

Worked examples

[Q1] Consider the sequence diagrams given below (from the handout [L4P1] Object-Oriented Programming: Intermediate Concepts)

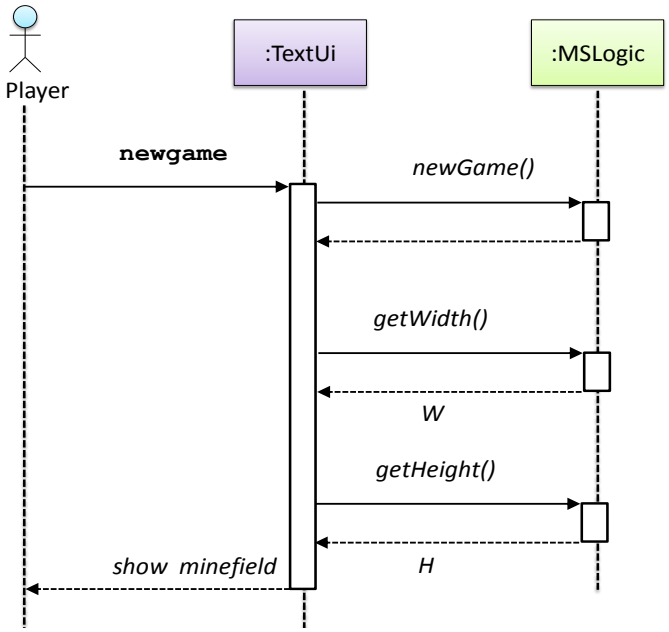


Figure 2. SD for 'new game' command

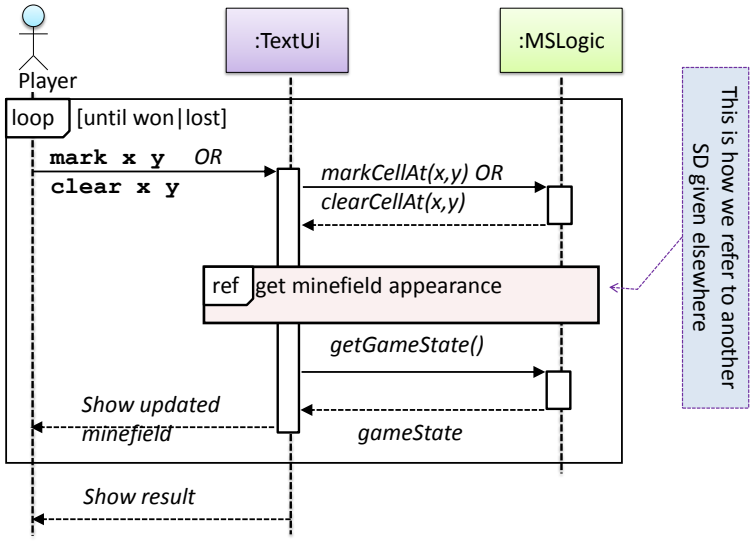


Figure 3. SD for commands 'mark' and 'clear'

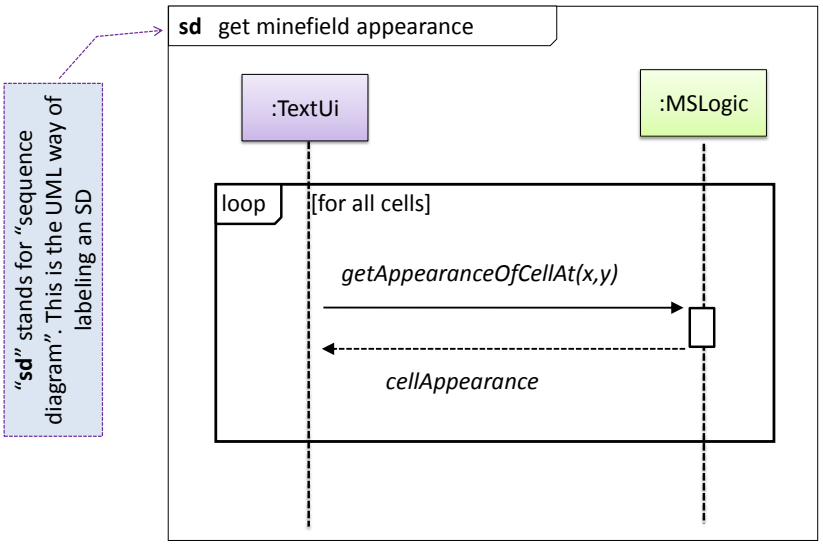


Figure 4. SD for 'get minefield appearance'

Here is the MSLogic API discovered by those diagrams:

```
newGame(): void
getWidth():int
getHeight():int
clearCellAt(int x, int y)
markCellAt(int x, int y)
getGameState() :GAME_STATE
getAppearanceOfCellAt(int x, int y): CELL_APPEARANCE
```

Modify those diagrams to accommodate the following features.

Feature id: tolerate

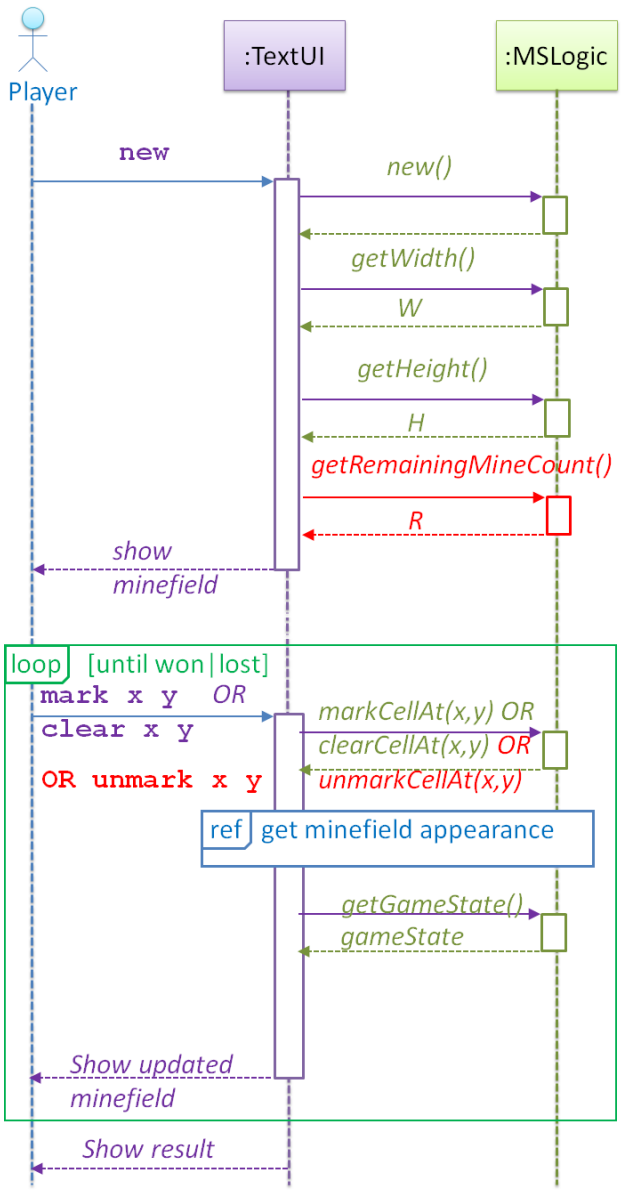
Description: Marking a cell incorrectly is tolerated as long as the number of cells marked does not exceed the total number of mines.

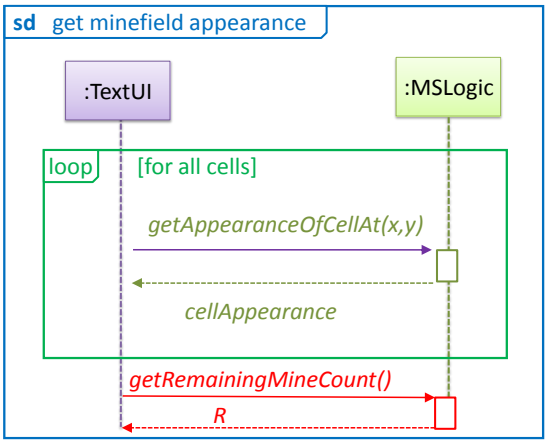
Depends on: show_remaining
…

Feature id: show_remaining
Description: The game shows the remaining number of mines during the game play. The number is calculated as (total mines – marked cells).

Feature id: unmark
Description: Marked cells can be unmarked, turning them back to hidden cells.

[A1]





Add to MSLogic API -> getRemainingMineCount():int, unmarkCellAt(int, int)

Note how the feature 'tolerate' does not have any effect on the three diagrams.

[Q2]

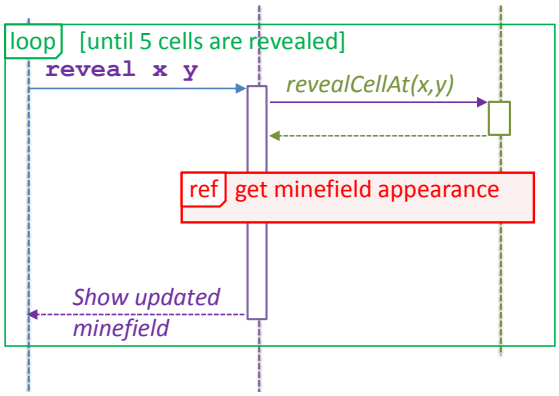
Consider the sequence diagrams in Q1. Show the modified versions of them, if any, after accommodating the following feature.

Feature id: standing_ground

Description: At the beginning of the game, the player chooses five cells to be revealed without penalty. This is done one cell at a time. If the cell so selected is mined, it will be marked automatically. The objective is to give some "standing ground" to the player from which he/she can deduce remaining cells. The player cannot mark or clear cells until the standing ground is selected.

[A2]

Insert this fragment between SD fragments shown below, between Figure 2 and Figure 3.



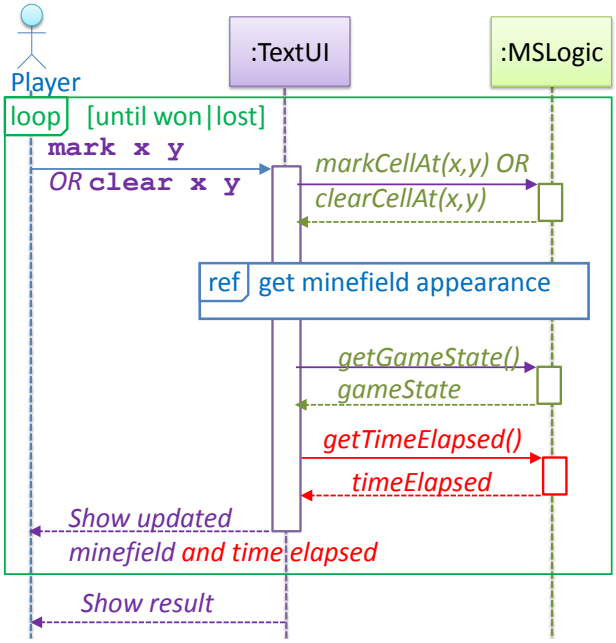
[Q3]

Same as the previous question, but for the feature given below:

Feature id: timing

Description: The game keeps track of the total time spent on a game. The counting starts from the moment the first cell is cleared or marked and stops when the game is won or lost. Time elapsed is shown to the player after every mark/clear operation.

[A3]

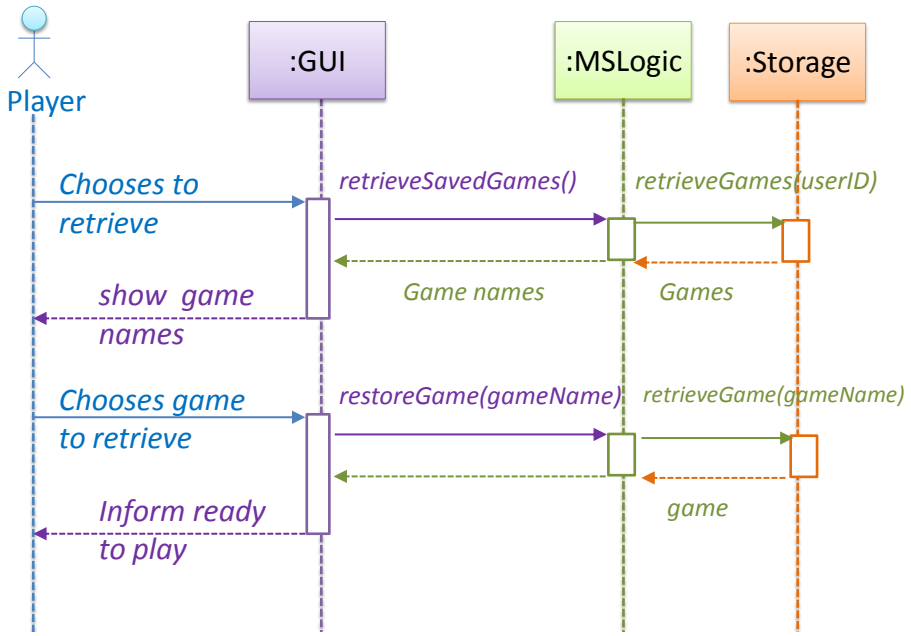


[Q4]

Draw the sequence diagram for the following use case. Include components: GUI, MSLogic, Storage. Here, we assume that Minesweeper allows saving and retrieving games. In addition, assume that the player is using the GUI. What are the API operations you discovered?

> Use case: 02 – retrieve game
> Actors: Player
> MSS:
> 1. Player requests to retrieve saved game.
> 2. Minesweeper shows a list of saved game by the same player.
> 3. Player chooses one of the games.
> 4. Minesweeper fetches the game from storage and informs the use it is ready to be played.
> Use case ends.

[A4]



MSLogic API (partial):

- retrieveSavedGames():String[]
  Overview: Used for retrieving all previous games saved by the current player.
  Returns: names of all games saved by the current player, sorted by the last modified date, most recent game appears first.
  Preconditions: none
  Postconditions: none

- restoreGame(String gameName):void
  Overview: Used to restore a game saved.
  Preconditions: the same player has previously saved a game under the gameName.
  Postconditions: the game is loaded and ready to be continued.

Storage API (partial):

- retrieveGames(String userID):Game[]
  …
- retrieveGame(String gameName):Game

--- End of handout ---