# [Handout for L4P1]
# The View from the Top: Architecture

After deciding *what* we should build (i.e establishing requirements and deciding system features), it is time to decide *how* we are going to build it (i.e. internal details of the implementation). The former (i.e. deciding what to build) is usually called the *analysis* phase while the latter (i.e. deciding how to build it) is usually called the *design* phase. However, there is no clear line to mark the end of one phase and start of the other. Often, a decision taken in analysis phase of one project could be taken in design phase of another project. For example, a decision like choice of development platform to be used for implementation of a system can be taken in the analysis or in the early design phase.

In simple cases, we can simply have a mental image of how we are going to structure our program and just dive into coding, without creating a design. In fact, code is a very detailed design. i.e. code is the 'blueprint' that will be compiled into the executable product. Sometimes, coding is the most efficient way to design. Therefore, 'design' does not necessarily mean drawing numerous diagrams. However, 'design by coding' does not scale up to larger, more complex, multi-person projects.

Design phase is generally divided into two levels – *preliminary* (also called high-level design), and *detailed* (also called low level design). In the high-level design, designers take decisions considering overall system, i.e. decisions here affect the system as a whole. High-level description of the overall system includes the top-level structure of subsystems and the roles and interactions of these subsystems. In detailed design, designers work at the module or sub-system level.

## Software architecture

Software architecture shows the overall organizations of the system. We can view it as a very high-level design. It should be technically viable, agreed upon, simple enough structure that is well understood by everyone in the development team, and that forms the basis for 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]*

Usually, architecture is developed by the *software architect.* A software architect provides the technical vision of the system and takes high-level (i.e. architecture-level) technical decisions of 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 3 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 tester used for testing the game logic

MSLogic: computation and logic of the game

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

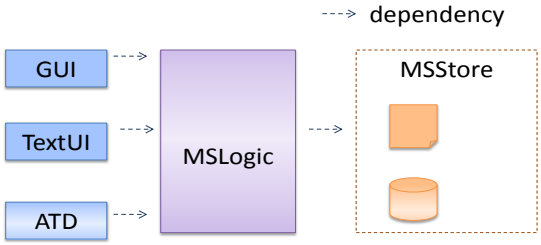



**Figure 3. A possible architecture for Minesweeper**

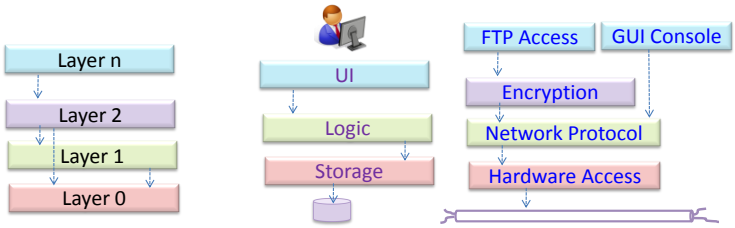
## Architectural styles

Given next are some examples of common architectural styles.

### The n-tier architectural style

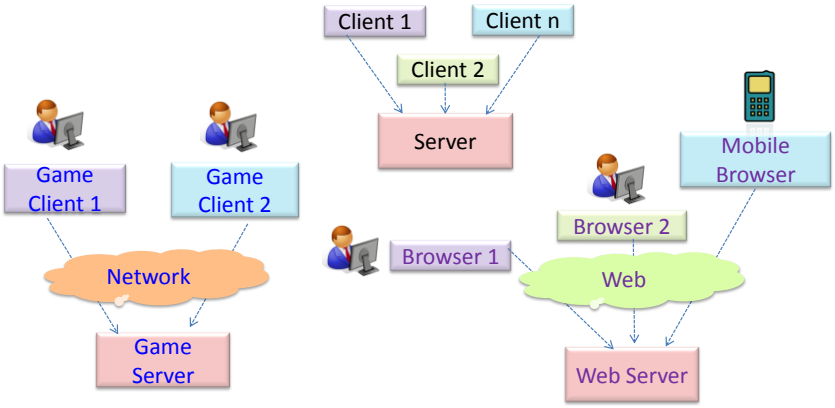
Main characteristic of an n-tier architectural style is that higher layers make use of services provided by lower layers. Lower layers are independent of higher layers.

Operating systems and network communication software often use this style.



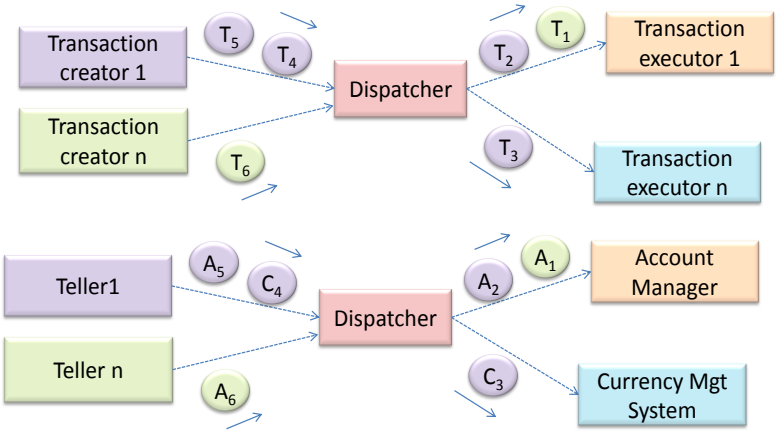

### The client-server architectural style

This is an architectural style for distributed applications. The main characteristic of client-server style is that at least one component plays the role of a server, and at least one client component accesses the server to make use of server's services.




**The transaction processing architectural style**

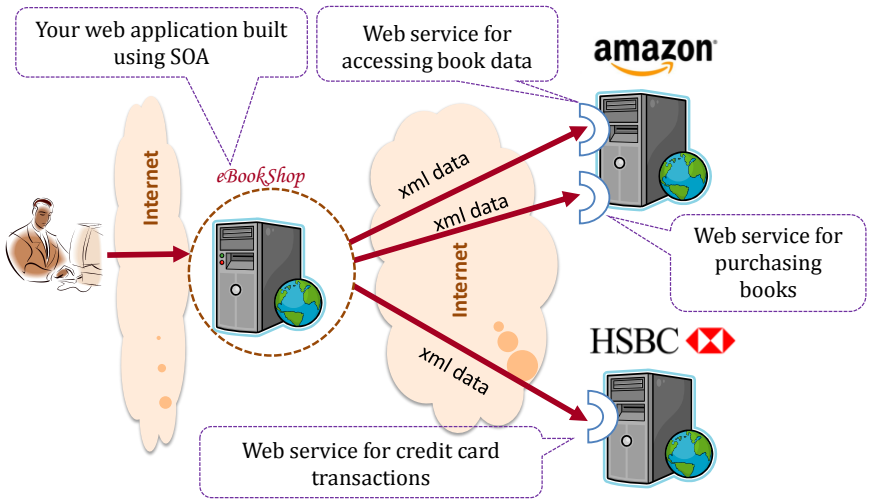
The main characteristic of a transaction processing architecture style is that the workload of the system is broken down to a number of *transactions*. These transactions are then given to a "dispatcher" that controls the execution of each transaction. Task queuing, ordering, "undo" etc. are handled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




**The Service-oriented architectural style**

*Service-oriented architecture* (SOA) is 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. *XML web services* is currently the common way to implement SOAs. Web services use web as the medium for the services to interact and XML as the language of communication between service providers and service users.

For example, assume that Amazon.com provides a web services to browse and buy merchandize from it and HSBC provides a web service for merchants to charge HSBC credit cards. Using these web services, you can write a web application that lets HSBC customers buy merchandize from Amazon and pay for them using HSBC credit cards. Because Amazon and HSBC services follow the SOA architecture, the eBookShop can reuse them to build an application on top of those services even if the three systems are using three different programming platforms.




**Other styles**

Other well-known architectural styles are 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 customize a style according to requirements. For example, an application can use a client-server architecture where the server component has several layers i.e. it uses the multi-layer architecture.

## Components and APIs

Minesweeper architecture given in Figure 3 shows the high-level components of the system and the anticipated dependencies between them. However, this architecture does not tell us how components communicate with each other i.e. which operations each component should provide so that they can work with each other to achieve the features of the system. In other words, we do not know the exact *interface* of these components. Here, the term 'interface' means the list of operations supported by a component and what each operation does. Once the interface is decided, implementation of various components can be done in parallel. 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 needs of its users) and well-documented. For an example, refer the API of the Java String component given here
http://download.oracle.com/javase/6/docs/api/java/lang/String.html

## Modeling component behavior using Sequence diagrams

As mentioned above, an API supports functionalities required from the component by its users. When designing a component, we first have to discover which functionalities are expected from that component.

Now, let us take the Minesweeper (MS for short) as an example to illustrate an approach to discover component APIs. A minimal MS should support at least the following use case. Note that this minimal MS does not tolerate any mistakes. The game is lost at the first mistake made by the player, even if the mistake is marking a cell as mined when it does not have a mine.

| |
|---|
| Use case: UC01 - Play Game<br>Actor: Player<br>1. Player starts a new game.<br>2. Minesweeper shows the minefield with all cells initially hidden.<br>3. Player marks or clears a hidden cell.<br>4. Minesweeper shows the updated minefield.<br>Steps 3-4 are repeated until the game is either won or lost.<br>5. Minesweeper shows the result.<br>Use case ends. |

Let us assume that user interacts with the system using a text UI. This interaction has already been captured in the use case. However, we need to specify it more formally (e.g., what exact commands will be used). Now, let us do that using a UML *sequence diagram*.

A UML sequence diagram captures the interactions between multiple components for a given scenario. Following is a sequence diagram (SD) that describes interactions between the player (an actor) and the TextUI component. Note that newgame and clear x y represent commands typed by the Player on the TextUI.

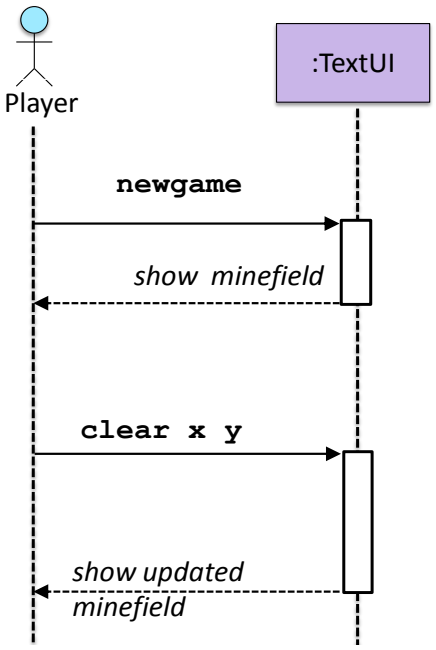



**Figure 4. SD for newgame and clear**

An explanation of the some basic elements of an SD is given below.

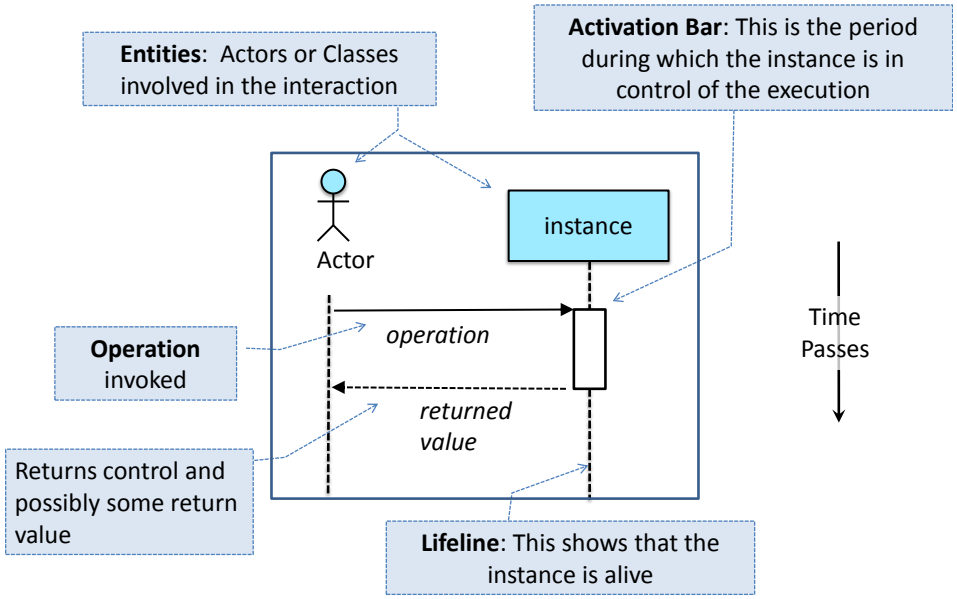



**Figure 5. Basic elements of an SD**

The notation ':TextUI' means 'an unnamed instance of the component TextUI'. If you had two TextUI instances in the diagram and wanted to distinguish between them you could name them 'textui1:TextUI' and 'textui2:TextUI'.

In the diagram below, an SD shows the interaction between the player and the TextUI for the full use case.

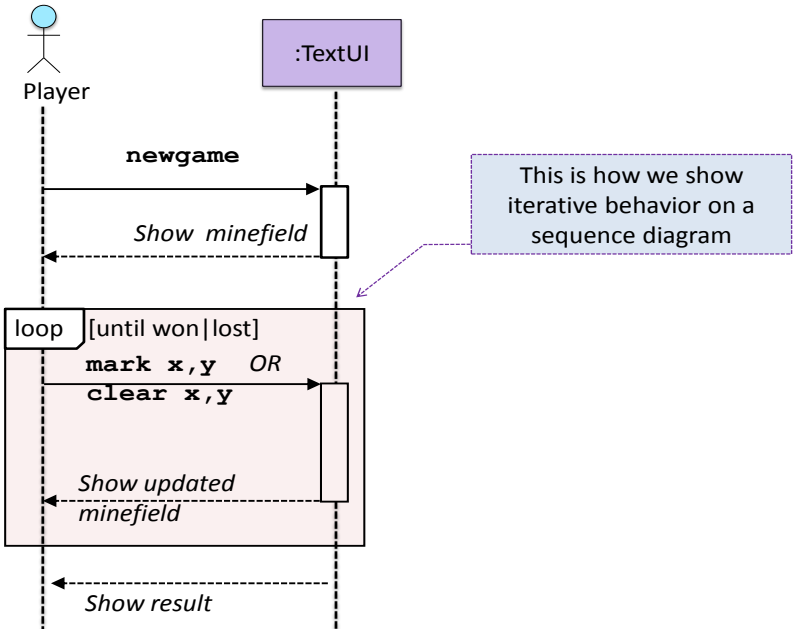



**Figure 6. Showing loops in an SD**

Now, how does the TextUI component carry out the requests it has received from player. It would need to interact with other components of the system which have been identified as part of the architecture. We had identified MSLogic component in the MS architecture. Given the role of the MSLogic (i.e. to control the game logic), it is natural

that TextUI will collaborate with MSLogic to fulfill the newgame request. Visually, this may be represented by extending the SD as below.

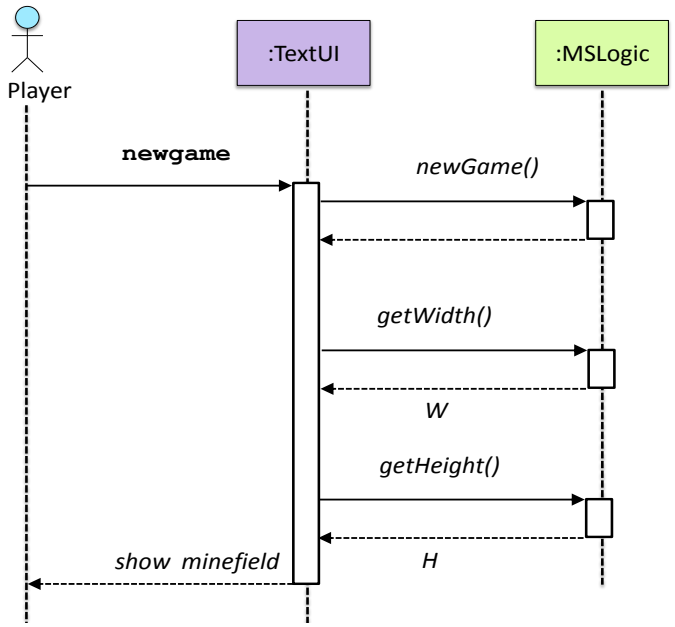



**Figure 7. TextUI and MSLogic interactions for newgame**

In the above diagram, we assume that the W and H (Width and Height of the minefield, respectively) are the only information TextUI requires to show the minefield to the Player. Note that there could be other ways of doing this.

MSLogic API discovered from this SD constitutes:

- newGame():void
- getWidth():int
- getHeight():int

Now, let us focus on the next task i.e. the mark or clear operations performed until the game is won or lost.

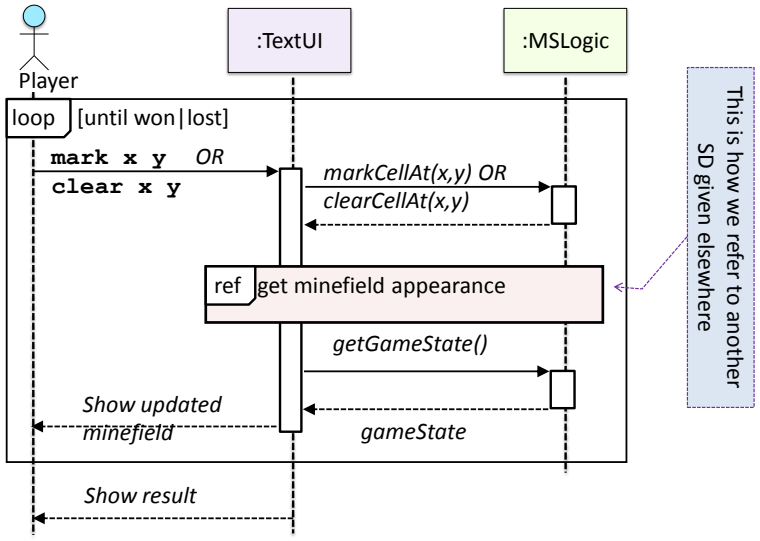



**Figure 8. Using a ref frame in an SD**

This interaction adds the following operations to the MSLogic API

- clearCellAt(int x, int y)
- markCellAt(int x, int y)
- getGameState() :GAME_STATE (GAME_STATE: READY, IN_PLAY, WON, LOST, …)

Note how we used a *ref frame* to omit a segment of the interaction so that we can show it separately in another diagram. That diagram is given below. It elaborates the retrieving of cell appearance from MSLogic.

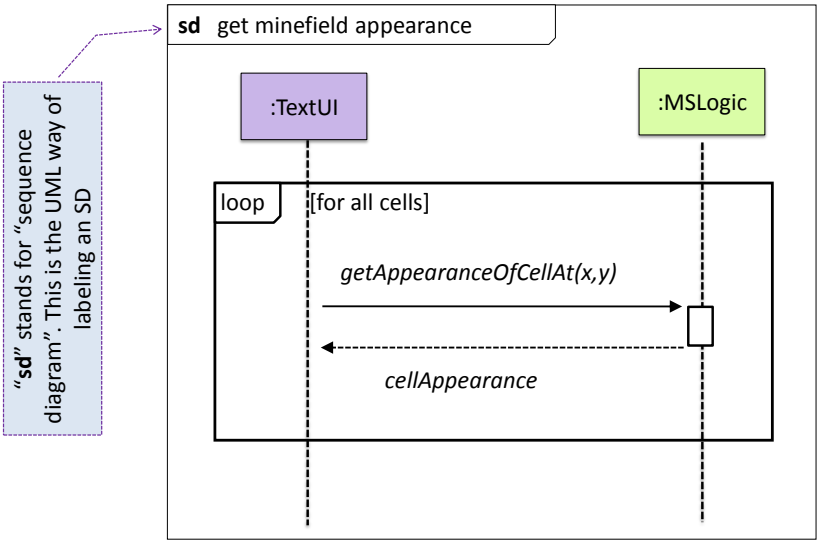



**Figure 9. The SD for the ref frame**

Accordingly, the following operation gets added to MSLogic API:

- getAppearanceOfCellAt(int,int):CELL_APPEARANCE (CELL_APPEARANCE: HIDDEN, ZERO, ONE, TWO, THREE, …, MARKED, INCORRECTLY_MARKED, …)

Note that in the above design, we do not let the TextUI access Cell objects directly. Instead, TextUI only gets information on what to show to the player, received as a value

of type CELL_APPEARANCE. Alternatively, we can pass the Cells or the entire Minefield directly to the TextUI.

To reduce clutter, we can omit activation bars and return arrows if that does not introduce ambiguities or loss of information. We can choose to use more informal operation descriptions such as given in the SD below, if the purpose is e.g. brainstorming and not the API specification.

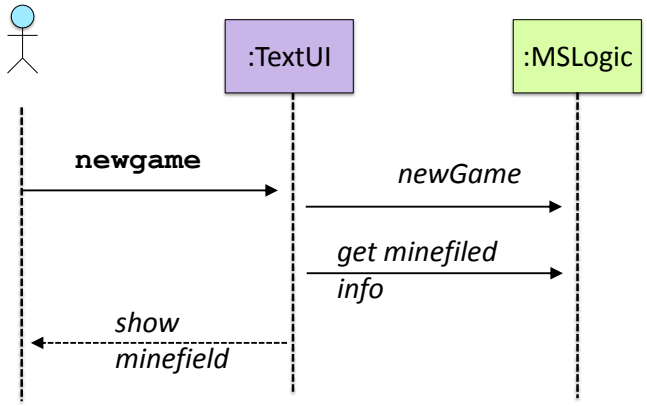



**Figure 10. A 'no frills' SD**

Here is the full MSLogic API we have discovered so far:

```
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
```

This is the case when Actor Player is interacting with the system using text UI. Additional operations (if any) required for the GUI and for the ATD can be discovered similarly.

Before we start coding, we can add some more details to increase the precision of the API. Such precision is important to avoid misunderstanding between the person who developes the component and those who develop other components that interact with this component.

**Operation**: *newGame(): void*
**Description**: Generates a new *WxH* minefield with *M* mines. Any existing minefield will be overwritten.
**Preconditions**: none.
**Postconditions**: A new minefield is created. Game state is READY.

Preconditions are the conditions that must be true before calling this operation. Postconditions describe the system *after* the operation is complete. Note that post conditions do not say what happens *during* the operation. Here is another example:

> **Operation**: *clearCellAt(int x, int y): void*
> **Description**: Records the cell at x,y as cleared.
> **Parameters**: x, y coordinates of the cell
> **Preconditions**: game state is READY or IN_PLAY. x and y are in 0..(H-1) and 0..(W-1), respectively.
> **Postconditions**:
> Cell at x,y changes state to CLEARED or INCORRECTLY_CLEARED
> Game state changes to IN_PLAY, WON or LOST as appropriate.

Other operations can be specified similarly. Note that preconditions and postconditions can be described as part of the operation 'description' instead of stating them separately. That is the approach taken by Java API descriptions.

There are several more useful SD notations not explained in this handout. They will be covered in a separate handout.

## Different approaches to design

### Top down vs bottom up

In the approach we are following in this handout, we started from very high-level by taking the system as one big black box and then broke it into a handful of smaller components. We can continue this approach for the remainder of the design process. That is, break each component into a small number of sub components, and so on until we have the complete detailed design. This approach is called *top-down design*. When using top down design, we do not have to worry about low level details until much later in the design and 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, we start with lower level details (e.g. data structures, storage formats, functions etc.) and progressively group them together to create bigger components. This is often used when the system is small or when we are building a variation of a product we've built before and we already have a large collection of reusable assets from the previous product that we can use in the new product.

### Agile vs full-design-up-front

Another aspect of design is how much of it should be done before starting the coding phase. In the industry, there are two schools of thought on this:

- *Agile-design* camp thinks it is 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 point trying to cater for all of them from the very beginning.

- *Full-design-up-front* camp thinks it should be designed fully 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

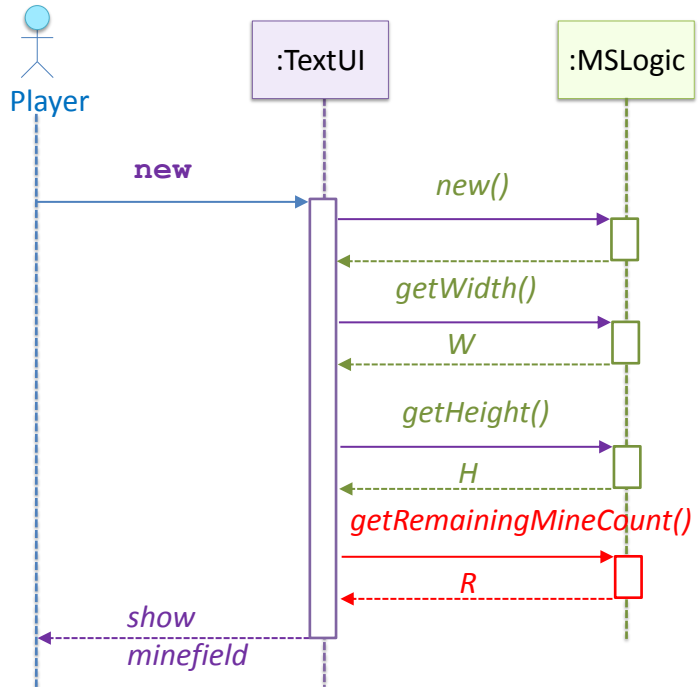
**[Q1]** Modify Figure 7, Figure 8, and Figure 9 and the MSLogic API to accommodate the following features.

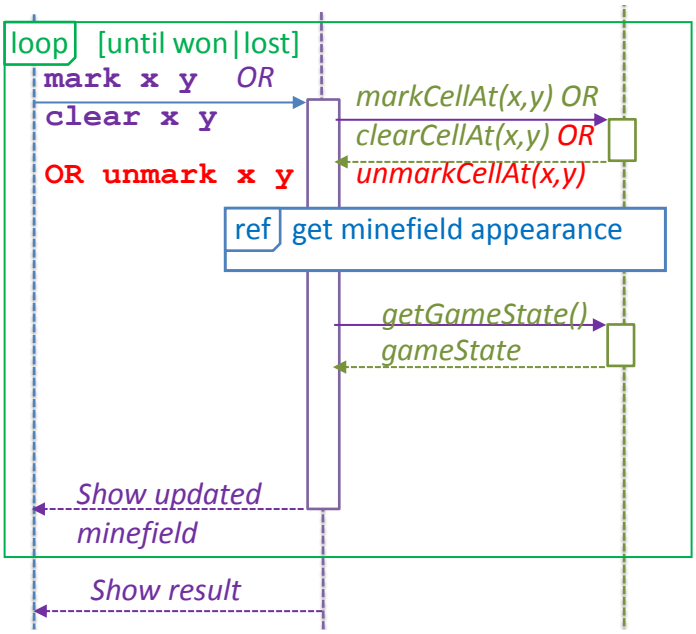
Feature id: tolerate
Description: Marking a cell incorrectly is tolerated as long as the number of cells marked does not exceed the total number of mines.
Depends on: show_remaining
...

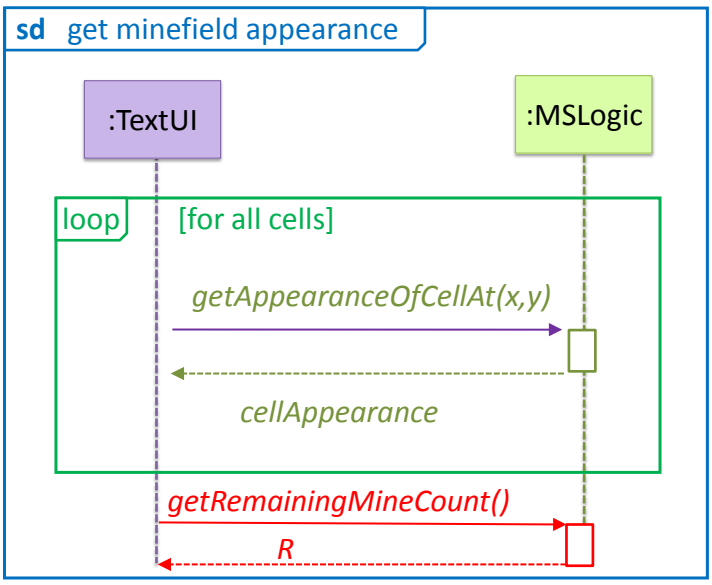
Feature id: show_remaining
Description: The game shows the remaining number of mines during the game play. The number is calculated as (total mines – marked cells).

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.

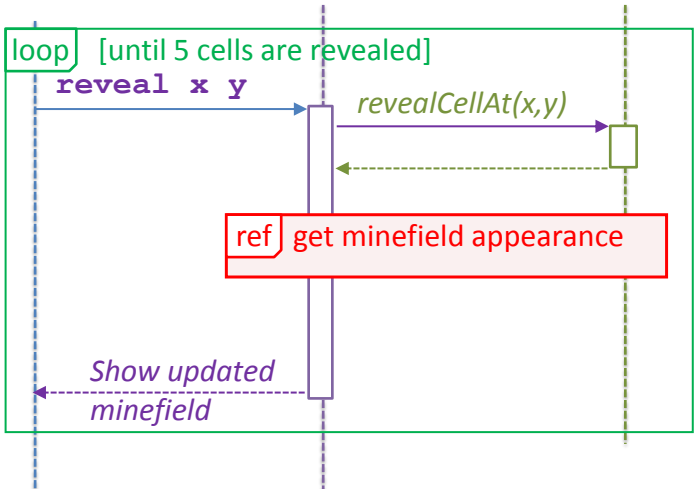
**[Q2]**
Consider sequence diagrams in the handout. Show the modified versions of them, if any, after accommodating the following feature.

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 in Figure 7 and Figure 8

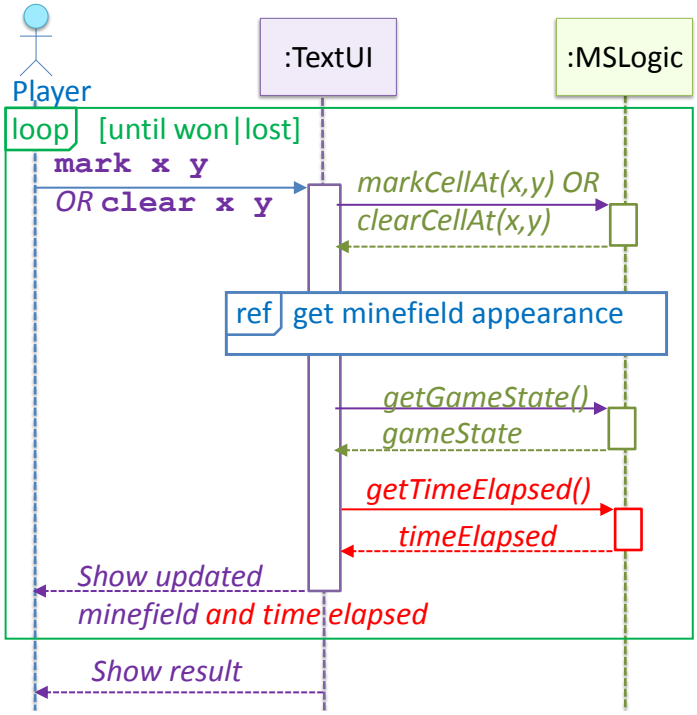



**[Q3]**

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]**

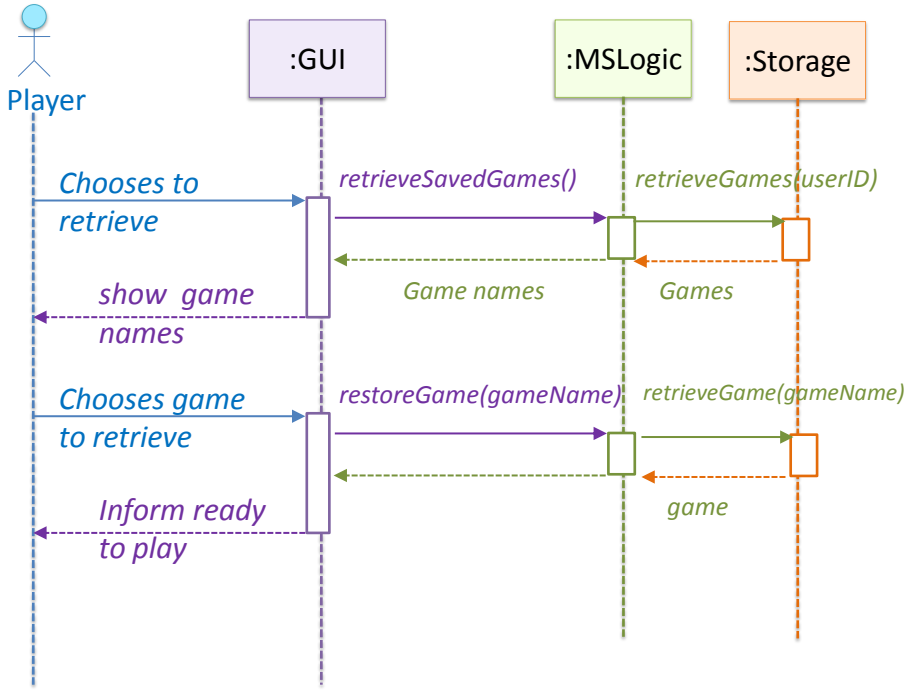


**[Q4]**

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 Document---