**איפיון QDA**

# High Level

## כללי

בשימוש במונחים ספציפיים כמו class וobject הכוונה היא לשימוש הרחב שלהם בעולם התכנות ולאו דווקא שימומש ע"י class בc וכו'.

בגדול – בQDA – כמעט - כל דבר זה Unit, כל מה שקורה זה Action והכל נכתב לLog.

## Qdb

Qdb הוא לא Action וגם לא Unit. Qdb הוא העולם של QDA.

* Qdb מכיל **Log** של כל מה שקרה בעולם הספציפי הזה.
* Qdb מכיל **unit** מיוחד שנקרא **UnitsManager**.
* Qdb מייצא ממשק להוספת פעולה חדשה ל**Log** ולביצוע הפעולה.
* Qdb מייצא את ממשק ה**Log**.
* Qdb מייצא את ממשק ה**UnitManager** (שבתורו מייצא את ממשקי קריאת הנתונים של כל ה**unitי**ם שבתוכו).
* Qdb מסתנכרן אל מול הstate הנוכחי של ה**Log** ע"י החלת הפעולות הנדרשות (או הrevert שלהן) על הunitים הקיימים.
* Qdb ריק זה Qdb שלא בוצע בו שום **Action** – ז"א עם **Log** ריק. בQdb כזה יש **Unit** אחד מסוג **UnitsManager** שמזוהה ע"י הGid 0.

## QDAClient

יש הרבה סוגים של **client**ים. למשל, client אינטרנטי, וclient מקומי. בסופו של דבר הclient מתממשק לQdb כלשהו (או יותר, אם הוא ממש רוצה) ומספק למשתמש גישה לQdb. בנוסף, הוא מאפשר גישה לנתונים של ה**unit**ים הקיימים באותו העולם. ע"מ לגשת לנתונים של**unit** ה**QDAClient** צריך להכיר את סוג המידע שנמצא ב**unit** (סוג המידע שנמצא בunit מיוצא גם ע"י **unit -> UnitsManager -> Qdb**).

## Unit

המבנה הבסיסי בQDA זה unit (יחידה). יש הרבה סוגים של יחידות, וניתן שיהיו קיימות מספר יחידות מאותו הסוג.

* כל יחידה מכילה נתונים שניתן לנהל אותם ע"פ סוג היחידה.
* ניהול הנתונים נעשה בלעדית באמצעות **פעולות (Action)**.
* כל unit מייצא רשימה של **Action**ים שהוא תומך בהם.
* כל סוג של יחידה מייצא פונקציות לגישה לנתונים שלו.
* מן הסתם, יש יחידות שתלויות ביחידות אחרות ע"מ לבצע את הפעולות שלהן.

## Action

Action זה פעולה שיכולה להתבצע ב**Qdb** כלשהו.

* כל פעולה מיוחסת ל**unit** מסוים והיא נכנסת ל**Log**.
* לכל פעולה חייבת להיות פעולה הופכית.
* *אין פעולות שמשפיעות על יותר מ****unit*** *אחד. {זו החלטה שאמורה לפשט. מקווה שהיא סבבה}.*
* **אוביקט פעולה** הוא דבר מאוד בסיסי וחסר לוגיקה משמעותית. מי שמבצע את הפעולה בפועל הוא הunit. (הunit מספק ממשק לביצוע פעולות – שמקבל את הפעולה בתור קלט).
* לכל Action יש Id.

# Workers

Workers זה מושג אבסטרקטי שמתאר ישויות שעובודות על Unitים ומבצעים Actionים עליהם. למשל, היוזר שפורש הוא Worker וכל פעולה שהוא עושה על הQdb מבוצעת באמצעות Actionים על Unitים. Workerים נוספים זה כל הכלים האוטומטיים שמבצעים הרבה פעולות על הQdb. למשל, יש AddressableUnit

***דוגמא:***

יש **Unit** שמתארקובץ בינארי כמו שהוא מופיע על הדיסק. נגיד Unit מסוג **UnitAddessable**. הunit הזה מחזיק data בינארי לפי כתובות. נניח שטוענים לUnit הזה קובץ elf מהדיסק. אז בUnit הזה הוא מופיע as is.  
עכשיו, יש ElfLoader – שהוא Worker. הוא יוצר עוד UnitAddessable (ע"י ביצוע Action על הUnitsManager) וממפה את הבתים מה UnitAddessable של "הקובץ על הדיסק" לUnitAddessable של "הקובץ טעון". אח"כ, יש Worker שיוצר Unit חדש, מסוג DisassemblyUnit, שמשתמש בUnitAddessable "הקובץ הטעון" ע"מ להכין Disassembly של הבינארי. וכו'.

# Detail Design

The detail design is the API of the major classes. The API declared as C++ functions. The implementation can be in any language.

## UnitTypeId

This is enum decalred as:  
**enum UnitTypeId;**

This enum contains unique id for every Unit subclass.

## ActionTypeId

This enum contains unique id for every Action subclass.

## Action

An abstract class declared as:  
**abstract class Action;**

**Action Action::Reverse();**Create a reverse Action object.

## Unit

An abstract class. Declared as:  
**abstract class Unit;**

**UnitTypeId Unit::GetUnitTypeId()**Returns a unique number that represents the unit type.

**List<ActionTypeId> Unit::GetActionTypes()**  
Returns a list of **ActionClass**es that the specific unit can handle.

**Status Unit::Act(const Action action&)**Perform an action on the unit.

## Log

The Log is the class that maintains all of the actions that was done in the **Qdb**.  
The Log contains **LogRecord**s.  
The log contains meta-data to maintain the current state of the log.  
The log should be implemented as tree to maintain branches.  
Every **LogRecord** should contain pair of unit\_id and action.

## UnitsManager

**Unit** subclass. Declared as:

**class UnitsManager:Unit**

This class is the main unit that the **Qdb** contains. It manages units.  
Every **Qdb** must have one and only one **UnitsManager** object.

The **UnitsManager** contains the whole units that exists in the Qdb. It responsible to create and delete units.

The **UnitsManager** can handle the following **action**s:

1. **ActionCreateUnit**

Creates an empty unnamed unit identified as unit\_id from a given unit\_type\_id.

The unit\_id have to be the next available unit\_id

1. **ActionDeleteUnit**

Delete an **empty** unnamed unit by unit\_id. A unit\_id unit from type unit\_type\_id has to be exists in the UnitsManager. There must not be units that depend on the unit to be deleted.

1. **ActionSetUnitName**

Change the unit\_id name from the old\_name to the new\_name.

1. **ActionAddUnitDependecy**

Mark that the unit depends on another unit.

1. **ActionRemoveUnitDependecy**

Remove the mark that the unit depends on another unit.

## UnitAddressable

This **Unit** contains advanced addressable binary data.  
Binary data is represents by the **Bytes** class.  
Every byte in this unit gets his value in that way:

1. If the byte has assigned value (**Regular** or **Uninitilize**), this value is returned.
2. If the byte has map source, the value from the source unit is returned.
3. Uninitialized byte is returned.

List<Pair<uint64,uint64>> UnitAddressable::GetRanges();  
Returns the valid addresses ranges (list of pairs of **start** and **size**).

Bytes UnitAddressable::GetBytes(uint64 address, uint64 size);  
Returns the bytes.  
The whole addresses have to be in range.

**Actions:**

1. **ActionAddRange(uint64 address, uint64 size);**Add range full with uninitialized data.
2. **ActionRemoveRange(uint64 address, uint64 size);**Remove range that was full with uninitialized data.
3. **ActionSetMap(uint64 this\_address, uint64 other\_address, uint64 size, Gid unit\_id);**Set the map source of the data.
4. **ActionUnsetMap(uint64 this\_address, uint64 other\_address, uint64 size, Gid unit\_id);**Unset the reference of the data.
5. **ActionSetData(uint64 address, Bytes old\_data, Bytes new\_data);**

## UnitDisassembly

This unit represents a disassembly of UnitAddressable.

The disassembly unit is coupled with a **UnitAddressable.** That means that the addressable ranges are the same as the coupled UnitAddressable.

An empty disassembly unit starts with all the data in **undefined** state. Every address can be assigned to be **Data** or **Instruction**. In order to make address to be **Data** or **Instruction**, all of the bytes that are part of the definition has to be in **undefined** state. **Data** addresses can be **byte, short, dword, qword, string, struct, etc.** Every defined address can be set with properties that marks the changes of this address from the default **Data/Instruction** state. (For example changes in the **Data** sign, or in the **Instruction** operand behavior to be offset).

**Actions:**

ActionCoupleWithUnitAddressable(Gid unit\_id);  
Couple new UnitDisassembly with an addressable unit. Can be used once per disassembly unit.

ActionUnCoupleWithUnitAddressable(Gid unit\_id);  
Uncouple.

**ActionDefineData(uint64 address)**Make Undefined address to be defined as default **data.** (byte)

**ActionUndefineData(uint64 address)**Reverse ActionDefineData

**ActionDefineInstruction(uint64 address)**Make undefined address to be defined as default **instruction**.

**ActionUndefineInstruction(uint64 address)**Reverse ActionDefineInstruction.

**ActionSetDataType(uint64 address, DataType old\_type, DataType new\_type);**

**ActionSetInstructionOverride(uint64 address, Instruction old\_override, Instruction new\_override);**

**ActionSetOperandProperty(uint64 address, int operand\_num, OperandProperty old\_property, OperandProperty new\_property);**

## Action sub classes pairs with their ActionTypeId (action and reverse action)

1. **ActionCreateUnit(UnitTypeId unit\_type\_id, Gid unit\_id); // rev = 2**
2. **ActionDeleteUnit(UnitTypeId unit\_type\_id, Gid unit\_id); // rev = 1**
3. **ActionSetUnitName(Gid unit\_id, wstring old\_name &, wstring new\_name &); // rev = 3**

## Bytes

The bytes class is used in order to manage strings of bytes.

string Bytes::get\_data(); // return string of 8-bit bytes that is the data itself.  
string Bytes::get\_status();   
return string of 8-bit bytes that represent the bytes status for every bytes:

1. **Regular**. In that case the byte from the data is used.
2. **Uninitialized**. “??”. In that case the byte-data has to be 0xff.
3. **Unused**. Represent a gap. In that case the byte-data has to be 0xff.
4. **Error**. Can’t get the byte value. In that case the byte-data has to be 0xff.