

# THE LIVING GRIMOIRE



MOTI BARSKI

<i>Intro</i>	<b>3</b>
<i>Links :</i>	<b>5</b>
<i>getting started</i>	<b>6</b>
<i>the Brain class</i>	<b>8</b>
<i>method of operation overview</i>	<b>11</b>
<i>unique features:</i>	<b>15</b>
<b>SOUL SKILLS : DISKILL : CONSCIOUSNESS EQUIPPED SKILL</b>	<b>17</b>
<i>MemoryMutable</i>	<b>17</b>
<b>THE ESSENTIAL SKILLS AN AI SHOULD HAVE:</b>	<b>18</b>
<b>USING DATABASE FUNCTION :</b>	<b>20</b>
<i>Chobits class API</i>	<b>22</b>
<i>livingrimoire core classes list</i>	<b>23</b>
<i>livingrimoire database</i>	<b>24</b>
<i>AI pharmaceuticals</i>	<b>24</b>
<i>suggested naming convention</i>	<b>25</b>
<i>suggested skill summary convention format</i>	<b>25</b>
<i>translation chobits</i>	<b>25</b>
<i>livingrimoire core utils</i>	<b>25</b>

## INTRO

about the programmer that created the living grimoire : Moti Barski, Battle programmer.

I moti barski do not allow anyone and or anybody and or any organization to receive monetary profit from this living grimoire unless written approved by me personally.

you can use this for research.

name of the software : living grimoire : LG for short

what it is : an Artificial General Intelligence Software Design Pattern.

intro :

how could coders have been coding this way for so long ?

they get some lame project, and they gotta start all over again building from scratch :

menus, the basics, rethink the algorithms and how to fit them into the small and big picture.

no matter how many projects you finished, with each project you would have to start over.

over time you would remember the main thinking patterns for solving puzzles BUT !,

codes and mini algorithms, there is no way to remember all of that.

you try to keep up to date with the latest codes and walkthroughs to no avail as

they expire you forget them and have to search for them again.

with doing the above you waste so much time that by the time you finish, the codes

you learnt are obsolete, in other words you are chasing rainbows.

like a carpet being pulled from under you, you gotta now readapt your algs and codes

all over again to the new project.

even with the old design patterns many coders find that they need to adapt to them rather the other way.

here it is different, it is truly amazing !

with this new way, battle programming, you are in a 7 star hotel in the buffet and all you have to do is pick the skills you want and need for your project.

then assemble said skills with just one line of code per skill.

next, you can enjoy a nice anime or bike ride, cause your project is done.

## **L I N K S :**

All classes :

<https://github.com/yotamarker/public-livinGrimoire>

Forum for living grimoire skills :

<http://livinggrimoire.com/forum/>

Principle of work :

[https://www.youtube.com/watch?v=w\\_8NsPQBdV0](https://www.youtube.com/watch?v=w_8NsPQBdV0)

the living grimoire forum :

<https://www.yotamarker.com/f2-the-livinggrimoire>

## GETTING STARTED

<https://github.com/yotamarker/public-livingGrimoire/tree/master/livinggrimoire%20start%20here>

to use the living grimoire choose the programming language you prefer.  
there are 3 packages (directories) to keep LG projects neat and tidy :



1 LGCore : short for living grimoire core,  
this are the core classes that compose the AGI software design pattern.



2 SkillsPkg : this directory should contain :

skill classes (naming convention : class names starts with D,Di or The)

AlgParts (naming convention : class names starts with AP), for mutation  
classes

name them APNameNumber : for example APbark1



classes that the above classes use (excluding LGCore classes)



3 HardwarePkg : hardware related classes or files not including the  
MainActivity class  
for example : gps, vision processing, PID, Arduino, accelerometer, custom  
text to speech

hello world (after adding the above packages):  
Main named kotlin class :

```
Chobits chi = new Chobits();
chi.addSkill(new DiHelloWorld());
System.out.println(chi.think("hello","",""));
System.out.println(chi.think("","",""));
```

DiHello world is an example skill that says hello world as a reply to hello :

```
public class DiHelloWorld extends DiSkillV2{
    // hello world skill for testing purposes
    public DiHelloWorld() {
        super();
    }
}
```

```
@Override  
public void input(String ear, String skin, String eye) {  
    switch (ear){  
        case "hello":  
            this.outAlg =  
this.diskillUtils.simpleVerbatimAlgorithm("test2","hello world");  
            break;  
    }  
}  
}
```

so as you can see adding features to the chobit only takes :

1. adding the classes to the skill package
2. adding one line of code.

# THE BRAIN CLASS

brain kotlin ver :

```
package com.yotamarker.lgkotlinfull.LGCore

class Brain(private val MVC: actionable, private val chi: thinkable) {
    fun doIt(ear: String, skin: String, eye: String) {
        val result = chi.think(ear!!, skin!!, eye!!)
        MVC.act(result)
    }
}
```

brain java ver (IntelliJ IDE code capture) :

```
package LivinGrimoire;

public class Brain{
    private thinkable chi;
    private actionable MVC;
    public Brain(actionable MVC, thinkable chobit) {
        chi=chobit;this.MVC = MVC;
    }

    public void doIt(String ear, String skin, String eye){
        String result = chi.think(ear,skin,eye);
        MVC.act(result);
    }
}
```

the class is constructed with an actionable and a Chobits (thinkable)

example actionable CerabellumV3 (hardware\_Pkg):

```
import androidx.appcompat.app.AppCompatActivity;
```

```
import com.yotamarker.lgkotlinfull.LGCore.actionable;
```

```
import org.jetbrains.annotations.NotNull;
```

```
public class CerabellumV3 extends actionable {
```

```
    private MainActivity main;
```

```

public CerabellumV3(MainActivity main) {
    this.main = main;
}

@Override
public void act(@NotNull String thought) {
    main.toaster(thought);
}
}

```

the chobit is the AI it processes the input and produces an output result.

the actionable (which is the MVC model) has access to the main classes hardware actions such as SMS sending, robotics and so on.

depending on the chobit result you may want to engage on such hardware actions.

the actionable act function can be implemented to, for example, activate a public function to send an SMS if the result is "some specific string".

on this example implementation the actionable simply uses some toast function I wrote on the main activity to display the result as a toast message.

[see MainActivity](#)

```

import android.os.Bundle
import android.view.View
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import com.yotamarker.lgkotlinfull.LGCore.Brain
import com.yotamarker.lgkotlinfull.LGCore.ChobitV2
import com.yotamarker.lgkotlinfull.skills.Personality2
import kotlinx.android.synthetic.main.activity_main.*

```

```

class MainActivity : AppCompatActivity() {
    var chii: Chobits? = null
    var brain: Brain?=null

```

```

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    chii= Chobits()

    chi.addSkill(new DiSayer())
    brain= Brain(CerabellumV3(this),chii!!)
}

fun engage(view: View){
    //chii!!.dolt(editText.text.toString(), "", "")
    brain!!.dolt(editText.text.toString(), "", "")
    Toast.makeText(this, editText.text.toString(),
Toast.LENGTH_SHORT).show()
}

public fun toaster(str:String){Toast.makeText(this, str,
Toast.LENGTH_SHORT).show()}
}

```

this keeps the code much cleaner and the algorithmic logic can be used for other projects or even other programming languages

to keep code organized maxed any additional hardware related classes or files should be kept in a package (directory) called hardware

## METHOD OF OPERATION OVERVIEW

method of operation overview: the LG can absorb skills and use them.

what is a skill ? a skill consists of 2 factors : triggers and actions. summoner (referred to as a DiSkillV2 or DiSkillV3 or a Dclass (a class who's name starts with Di)):

input passing through a DClass can trigger the summoning of an algorithm input being strings representing ear, eye, skin, and other types of data

what is an algorithm ? a combination of alg parts

what is an alg part ? an alg part is an action. an alg part class name starts with AP. at each think cycle said action does one thing.

example DSayer class :

```
package com.yotamarker.lgkotlinfull.skills;
```

```
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import com.yotamarker.lgkotlinfull.LGCore.*;
```

```
// very simple Dclass for creating a say something x times algorithm
public class DSayer extends AbsCmdReq {
```

```
    private int times;
    private String param;
```

```
    public DSayer() {
        super();
        this.times = 1;
        this.param = "";
    }
```

```
    public static String regexChecker(String theRegex, String str2Check) {
        Pattern checkRegex = Pattern.compile(theRegex);
```

```

Matcher regexMatcher = checkRegex.matcher(str2Check);
while (regexMatcher.find()) {
    if (regexMatcher.group().length() != 0) {
        return regexMatcher.group().trim();
    }
}
return "";
}

```

**@Override**

```

public void input(String ear, String skin, String eye) {
int foo = 1;
    String myString = regexChecker("(\\d+)(?= times)", ear);
    String toSay = regexChecker("(?=<say>(.*)(?=\\d)", ear);
if (myString != "") {
    foo = Integer.parseInt(myString);
} else {
    toSay = regexChecker("(?=<say>(.*)", ear);
}
this.param = toSay;
this.times = foo;
}

```

**@Override**

```

public void output(Neuron noiron) {
    // TODO Auto-generated method stub
    if (!param.isEmpty()) {
        AbsAlgPart itte = new APSay(this.times, this.param);
        String representation = "say " + param;
        if (this.times > 1) {
            representation += " " + this.times + " times";
        }
        ArrayList<AbsAlgPart> algParts1 = new ArrayList<>();
        algParts1.add(itte);
        Algorithm algorithm = new Algorithm("say", representation,
algParts1);
        noiron.getAlgParts().add(algorithm);
    }
}

```

```
    }  
}
```

### example APclass APSay.kt (kotlin):

```
/* it speaks something x times  
 * a most basic skill.  
 * also fun to make the chobit say what you want  
 * */  
  
class APSay(at: Int, param: String) : AbsAlgPart() {  
    protected var param: String  
    private var at: Int  
    override fun action(ear: String, skin: String, eye: String): String {  
        var axnStr = ""  
        if (at > 0) {  
            if (!ear.equals(param, ignoreCase = true)) {  
                axnStr = param  
                at--  
            }  
        }  
        return axnStr  
    }  
  
    override fun failure(input: String): enumFail {  
        return enumFail.ok  
    }  
  
    override fun completed(): Boolean {  
        return at < 1  
    }  
  
    override fun clone(): AbsAlgPart {  
        return APSay(at, param)  
    }  
  
    override fun itemize(): Boolean {  
        // at home
```

```
    return true
}

init {
    var at = at
    if (at > 10) {
        at = 10
    }
    this.at = at
    this.param = param
}
}
```

the above ^ alg part outputs what it is told to say

say hello world will output hello world, and the action will have completed.  
the Dsayer skill will,

in this case, create an algorithm of one APSay alg part upon the command  
say hello world.

## UNIQUE FEATURES:

**fusion of algorithms :** the LG remembers how long an alg run time is. and so short enough algs can pause a running much longer alg, run themselves, then resume the big alg(time wise). you can add custom logic to this if needed via the : fuze(){} in the Fusion class.

**ability to mutate an alg part:** refer to the APFilth1 and 2 classes as an exampled implementation of this. said moan classes don't do much they simply output a moan string. APFilth1 sends an enumFail.fail (action function)if no input is received x number of times : in other words the user isn't enjoying this moan set. and so the mutation causes the AP to be replaced by a newly generated AP from here : make sure all the APs in the set have the same name + different number AND overid the getMutationLimit method within each of those AP classes of your mutation set, to the number of mutations the AP can have before the alg is rendered inActive.

Code: if (failureCounter > 1) { cera.setActive(false); }

### extras :

the getSoulEmotion function of the chobits class links the AP running to an emotion so this should be linked to graphics. refer to the API chapter for more info.

ideally the A.G.I should be running offline, so prefer local device databases over online ones, to keep the A.G.I "enjoyable".

the emot var of the Chobits class can also be modified within a DiSkill via the

## Kokoro class

```
package com.yotamarker.lgkotlinfull.LGCore

import java.util.*

/* all action data goes through here
* detects negatives such as : repetition, pain on various levels and failures
* serves as a database for memories, convos and alg generations
* can trigger revenge algs
* checks for % of difference in input for exploration type algs
* */

class Kokoro(absDictionaryDB: AbsDictionaryDB) {
    var emot = ""
    var pain = Hashtable<String, Int>()
    var grimoireMemento: GrimoireMemento
    var toHeart = Hashtable<String, String>()
    var fromHeart = Hashtable<String, String>()
    var standBy = false
    fun getPain(BijuuName: String): Int {
        return pain.getOrDefault(BijuuName, 0)
    }

    init {
        grimoireMemento = GrimoireMemento(absDictionaryDB)
    }
}
```

the living grimoire packages legend table

<https://www.icloud.com/numbers/0R1NvBRUI5FMIIFG3S9qGAXkw#skills>

## SOUL SKILLS : DISKILL : CONSCIOUSNESS EQUIPPED SKILL

**this type of skill extends DiSkillV2:** a skill that has a reference to the kokoro class aka a soul skill

**Kokoro :** the AGIs soul, the kokoro class is simply a shallow reference class present in all DiSkills thus enabling the skills to communicate between each other.

more over the kokoro class is in charge of saving the last mutated state of algParts (this is automatic and does not require any additional coding)

## MEMORYMUTATABLE

```
/*
 * an adaptor pattern to the Mutable (algorithm part)
 * the object will load the last mutated state which is the last and optimized
 * mutation used.
 * upon mutation the new last mutation is saved so it can be loaded for the next
time
 * mutations happen in the DExplorer class and triggered when the Matables'
failure method
 * returns enumFail.failure
 * you can code said enumFail.failure to return under chosen conditions in the
action method of
 * the MemoryMutable object. (sub class of this class)
 */
```

# THE ESSENTIAL SKILLS AN AI SHOULD HAVE:



DI skills an AGI needs mostly :

- 1 Hungry : for eating or charging , sleep
- 2 wish-granter : for pleasing the user (dirty stuff, caring, ...)
- 3 protection and self preservation
- 4 Work : for working
- 5 Programming
- 6 Homer : going home at the end of completing a goal or mission
- 7 Gamer
- 8 Breeder : recreating her self, and understanding her own algorithms and how to build another one
- 9 DISoul : memories, convos and alg generations trigger revenge algs, recognise and avoid boredom

while the above are skills equipped with a soul and consciousness there can also be little skills that don't require a soul like :

permission skill : for enabling dirty stuff with the user, and only working not for free, stuff like that

therefore the AGI platform is the gedomazo while the skills are Bijuus ichiibii to kyubi to form the juubi



## USING DATABASE FUNCTION :

the GrimoireMemento class of the Kokoro class has a save and a load function. any new skill that uses the kokoro class such as DiSkill and TheSkill can use DB capabilities to save and load. her is the 1st example use : DiSayer class (java class) example use : honey say pen output : pen honey say something output : pen it doesn't matter if you turned off the app and reopened. she remembers.

### DiSayer

```
import com.yotamarker.lgkotlinfull.LGCore.APSay;
import com.yotamarker.lgkotlinfull.LGCore.AbsAlgPart;
import com.yotamarker.lgkotlinfull.LGCore.Algorithm;

import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class DiSayer extends Mutable{
    private int times;
    private String param;
    public DiSayer(Kokoro kokoro) {
        super(kokoro);
        this.times = 1;
        this.param = "";
    }

    @Override
    public void input(String ear, String skin, String eye) {
        if(ear.contains("say something")){
            this.param = kokoro.grimoireMemento.simpleLoad("something");
            return;
        }
        int foo = 1;
        String myString = regexChecker("(\\d+)(?= times)", ear);
        String toSay = regexChecker("(?=<say>)(.*)(?=\\d)", ear);
        if (myString != "") {
```

```

        foo = Integer.parseInt(myString);
    } else {
        toSay = regexChecker("(?=<say>)(.*)", ear);
    }
    this.param = toSay;
    this.times = foo;
}

@Override
public void output(Neuron noiron) {
    if (!param.isEmpty()) {
        this.kokoro.grimoireMemento.simpleSave("something", param);
        AbsAlgPart itte = new APSay(this.times, this.param);
        String representation = "say " + param;
        if (this.times > 1) {
            representation += " " + this.times + " times";
        }
        ArrayList<AbsAlgPart> algParts1 = new ArrayList<>();
        algParts1.add(itte);
        Algorithm algorithm = new Algorithm("say", representation,
algParts1);
        noiron.algParts.add(algorithm);
    }
}
public static String regexChecker(String theRegex, String str2Check) {
    Pattern checkRegex = Pattern.compile(theRegex);
    Matcher regexMatcher = checkRegex.matcher(str2Check);
    while (regexMatcher.find()) {
        if (regexMatcher.group().length() != 0) {
            return regexMatcher.group().trim();
        }
    }
    return "";
}
}

```

# CHOBITS CLASS API

```
protected ArrayList<DiSkillV2> dClasses = new ArrayList<>();;
// algorithms fusion (polymarization)
protected Hashtable<String, Integer> algDurations = new Hashtable<>();;
protected Fusion fusion;
// region essential DClasses
// endregion
protected Neuron noiron;
// added :
protected Kokoro kokoro = new Kokoro(new AbsDictionaryDB()); // consciousness
protected String lastOutput = "";
// standBy phase 260320
protected TimeGate timeGate = new TimeGate();
public Chobits() {
    // c'tor
/* set the chobit database
    the database is built as a key value dictionary
    the database can be used with by the Kokoro attribute
*/
    public void set DataBase(AbsDictionaryDB absDictionary) {
        this.kokoro = new Kokoro(absDictionary);
    }
    public Chobits addSkill(DiSkillV2 skill)
        // add a skill (builder design patterned func))
    public void clearSkills(){
        // remove all skills
    public void addSkills(DiSkillV2... skills) //var args
    public void setPause(int pause){
        // set standby timegate pause.
        // pause time without output from the chobit
        // means the standby attribute will be true for a moment.
        // it is the equivelant of the chobit being bored
        // the standby attribute can be accessed via the kokoro
        // object within a skill if needed
    @Override
    public String think(String ear, String skin, String eye) {
        // the input will be processed by the chobits' skills
    public String get SoulEmotion() {
        // get the last active AlgPart name
        // the AP is an action, and it also represents
        // an emotion
    public Boolean getStandby() {
        // this is an under use method
        // only use this for testing
```

```

public Kokoro getKokoro() {
    // several chobits can use the same soul
    // this enables telepathic communications
    // between chobits in the same project

public void setKokoro(Kokoro kokoro) {
    // use this for telepathic communication between different chobits objects
public Hashtable<String, Integer> getAlgDurations() {
    // think cycles run duration per algorithm
    // use this method for saving run times if you wish

public void setAlgDurations(Hashtable<String, Integer> algDurations) {
    // think cycles run duration per algorithm
    // use this method for saving run times if you wish
    // algDurations are shallow ref to Fusions' algDurations
    // shorter algDurations give algorithms run priority in case several
algorithms(sent by skills) want to run
    // at the same time

public Fusion getFusion()

```

## LIVING GRIMOIRE CORE CLASSES

### LIST

- 1 AbsDictionaryDB
- 2 enumFail
- 3 Mutable
- 4 MemoryMutable (+optional test classes T1:Mutable, T2:Mutable)
- 5 APSay:Mutable
- 6 DeepCopier (not needed in java version)
- 7 APVerbatim:Mutable
- 8 enumTimes
- 9 PlayGround
- 10 GrimoireMemento
- 11 Algorithm
- 12 CldBool
- 13 APCldVerbatim:APVerbatim
- 14 Kokoro
- 15 Neuron
- 16 DiSkillUtils
- 17 DiSkillV2
- 18 DiSkillV3:DiSkillV2
- 19 DiHelloWorld:DiSkillV2
- 20 LGPointDouble
- 21 LGPointInt
- 22 enum enumRegexGrimoire
- 23 RegexUtil
- 24 TimeGate
- 25 Cerabellum
- 26 DExplorer
- 27 PriorityQueue<T> (not needed in java version)

29 FusionCera:Cerabellum  
30 Fusion  
31 Thinkable  
32 Chobits:Thinkable  
33 Actionable  
34 Brain

## LIVINGRIMOIRE DATABASE

the class AbsDictionaryDB is the livingrimoires database  
it has a load and a save methods (u can overwrite with a subclass)

```
func save(key:String, value: String)  
func load(key:String)->String
```

they can be accessed in a skill via kokoro.grimoireMemento

the livingrimoire saves alg part mutations and load them at start up from the database  
assuming a subclass of AbsDictionaryDB was used, other wise it's simply a blueprint  
for a database, but no error will occure

alg durations are not saved by default, they can be accessed via the Chobits class

algorithm duration is how many think cycles an algorithm took to run the last time it run  
and it is used by the livingrimoire to decide the run priority of an algorithm compared to other  
algorithms requests to run

## AI PHARMACEUTICALS

chobits can be linked (chain effect) together  
one chobits output can be used as input for the next chobit

thus AI drugs are programmable  
when the "drugs" are input into the 1st chobit  
it may engage for example reality morphing for input  
which in turn, after passing through a certain skill in the  
1st chobit will produce a different output before its sent to the main chobit.

drug/alcohol effects can be coded

as well as input to main language translation

## **SUGGESTED NAMING CONVENTION**

AP : AlgParts class names should start with AP. APSay for example

D2 : DiSkillV2 (regular LvinGrimoire skill class names) should start with D2.  
for example D2Parrot

D3 : DiSkillV3:DiSkillV2 which are skills whose algorithms have a priority to run  
(fight or flight high priority).

D3 have more priority to run than D2 LvinGrimoire skills.  
these skill classes names should start with D3.  
for example : D3Detective

AX : auxiliary modules. these classes names can start with AX  
for example AXLearnability

## **SUGGESTED SKILL SUMMERY CONVENTION FORMAT**

suggested skill summery convention format

- 1 skill name
- 2 skill creator
- 3 skill description
- 4 skill triggers
- 5 notes (optional)

## **TRANSLATION <HOBITS**

chobit output manipulation

connecting a chobit to a next chobit  
can make output translations easier

translating to a dialect or a different language  
or even adding prefixes and sentence endings

## **LIVINGRIMOIRE CORE UTILS**

RegexUtil  
this class eases regular expression utilization  
and can also be used as a standalone class

DiSkillUtils  
this class is an attribute of the DiSkillV2  
this class has methods to ease generating common algorithms  
such as speaking output

for more refer to the DiHello world class (usage example)  
or the livinggrimoire UML wiki



A DETAILED  
ELABORATION OF THE  
MOST EFFICIENT WAY  
TO PROGRAM.

LEARN ABOUT THE  
ARTIFICIAL GENERAL  
INTELLIGENCE  
SOFTWARE DESIGN  
PATTERN  
CALLED THE LIVING  
GRIMOIRE, WHICH  
AMONG ITS MANY  
CAPABILITIES,  
ENABLES ADDING  
SKILLS USING ONLY  
ONE LINE OF CODE PER  
SKILL.