Logic Engine case study – Expense Application System

(contact@witmate.com)

Abstract

This article explains the approaches and advantages of Logic Engine based system design and implementation with the case of an expense application system. We use a java Logic Engine named Witmate (http://www.witmate.com/) as the implementation tool. We ignored many details of this case to focus on the central idea of Logic Engine.

The Case

A company with hundreds employees wants to develop an expense application IT system replacing the current manual operations. Employees apply their expense of one year at the end of each year by these steps:

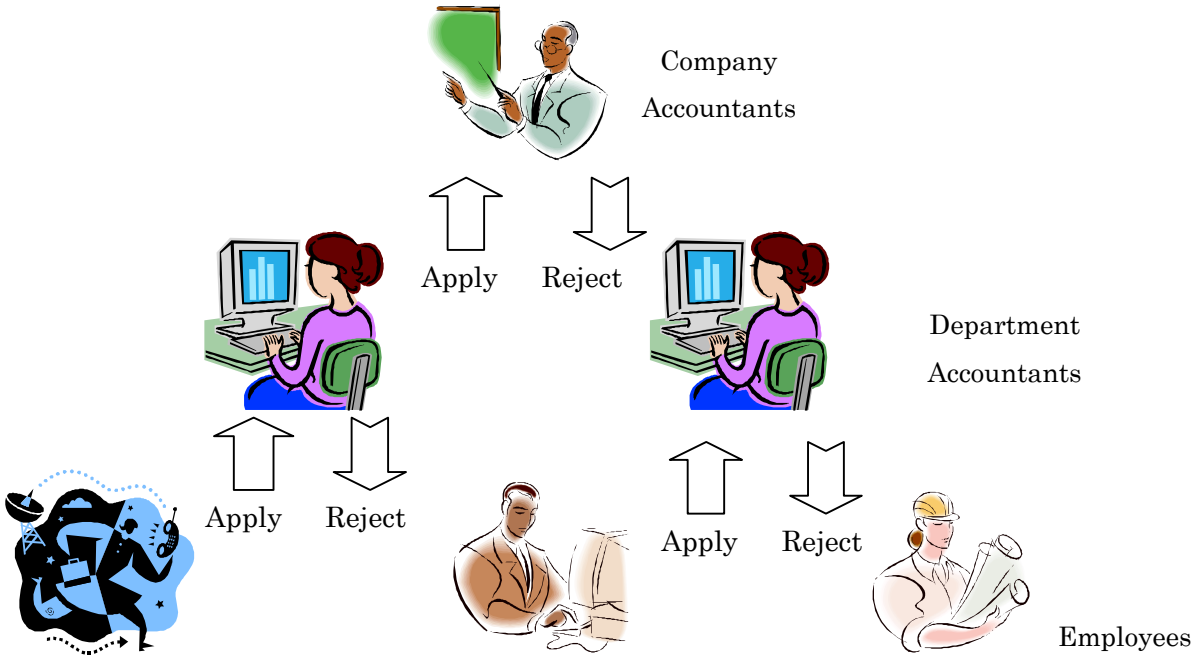
1. Employees fill out the application form, and send it to their department accountants.
2. Department accountants check these applications whether match the expense plan that is decided at the beginning of every year. The expense plan includes total expense number of whole department and limited numbers of different positions employees, and balancing logics. The balancing logics express what is the applicable balances among the items of expenses, for example:
   - The travel expense of one employee can not over $3000 in one year
   - If one employee's over sea calls fee over 50% of her total expense, her travel expense can not over $2000

   Department accountants will reject forms that are over the numbers or do not match the balancing logics to employees to refill, and report all fixed and correct forms to accountants of company.
3. Company accountants check all application forms of departments according to company level numbers and balancing logics

   Any wrong forms will be rejected.

As any account systems, the numbers and balancing logics are always changing from year to year.

This is the system diagram:



The major targets that the company develops the IT system are:

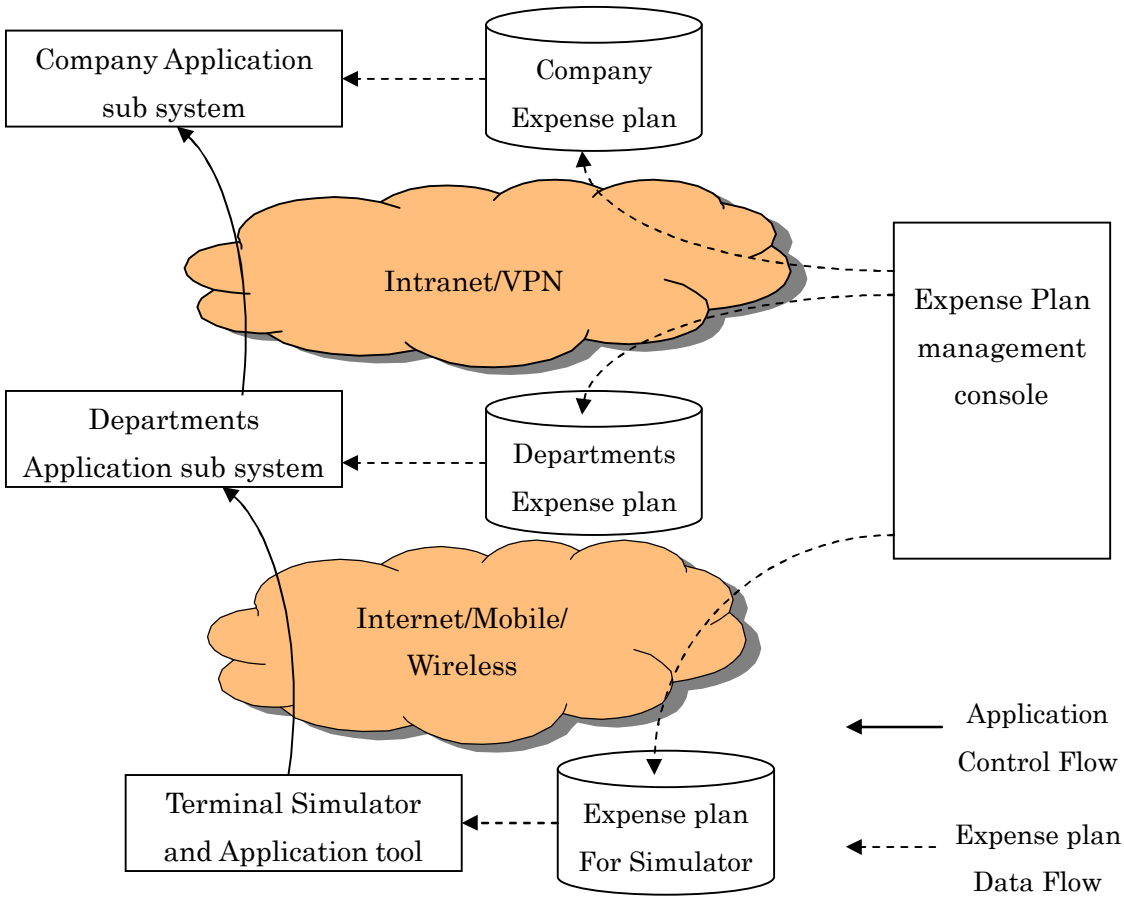
- Automatic the operation process to speed up operations.
- Decrease mistakes that are caused by year to year changes of balancing logics and government requirements.
- Decrease rejections to save time and cost.

IT infra status of this company is that most of the employees can get chance to use a client terminal. But the types of these terminals are very multiple, from Smart phones, PDAs, PCs to UNIX workstations.

System design

According to the multiple terminal types of customer's IT infra, we choose Java platform as the basic platform to develop most applications of this system.

Jumped over some steps, we got the basic system architecture:



To decrease mistakes caused by plan changes, an expense plan management console is required to control all plans centrally. The console also has the functionality to broadcast plans defined to sub systems.

To decrease rejections, a simulator is deployed into terminals. Employees will use it to simulate their applications on their terminal before actual application.

The next step of a logic engine based system design is clarifying what parts of system should be done by programmers and what parts can be managed by experts leveraging logic engine. In this case, experts can do:

- Expense plans definition and management
- Inputs and outputs definitions of expense plans

And programmers do:

- Sub systems implementation

- Database design and implementation
- Communication through Networks

Company expense plan, department expense plan and expense plan for simulator are defined by logic description languages which are saved in Relation Databases, files and Mobile device memories. So logic language format must fit for different devices and storages.

Expense plan need to be broadcast to sub systems and terminals through LAN, VPN and mobile/wireless networks.

General workflow of this system is:

1. Expense experts define expense plans and broadcast plans to sub systems and terminals by expense management console.
2. Employees input their expense numbers and simulate the application to check whether there are wrong numbers in their applications on their own terminals.
3. Employees apply their expenses to department sub system
4. Department accountants check their department employees' applications according to department expense plans, and apply summed applications to company sub system with Departments Application sub system.
5. Company accountants check all applications according to company expense plan, and get report of expense for company this year.

After collected requirements and got the basic design architecture, let's go to next step to implement this system with Logic engine Witmate.

System implementation

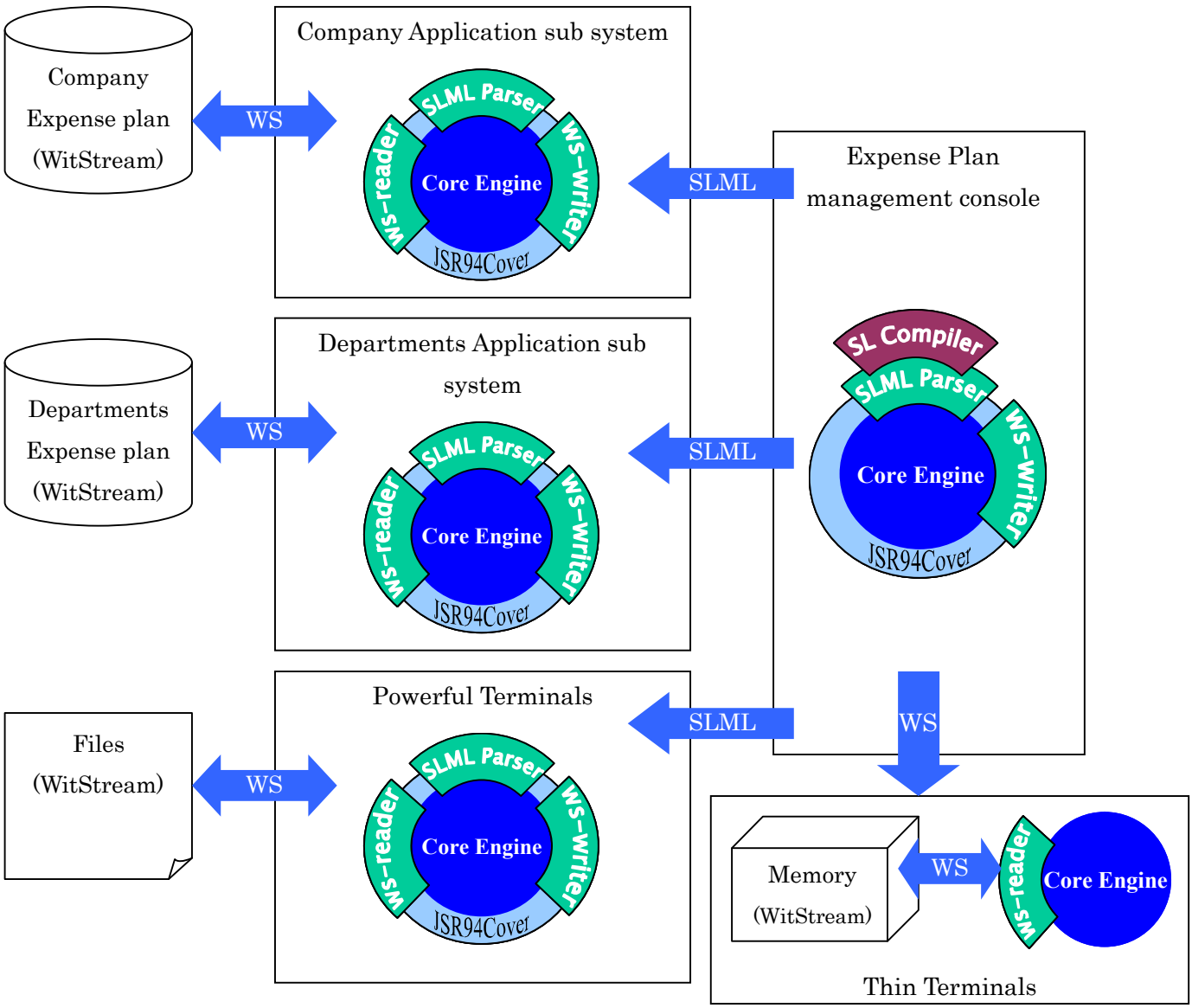
We use Witmate logic engine to implement this system, because Witmate is the only one java logic engine executable from j2me to j2se/j2ee to support multiple types of devices of this system, and the multiple logic formats feature of Witmate is the best solution of logics definition and broadcast through multiple type networks.

Expense plan management console is implemented as 2 editions. The first and simpler one is a Simple Language (SL) editor based edition. Experts will use a text editor to write and modify SL logic definition files. The second and more complex edition is a GUI based logic management tool and the generation of the GUI tool is logic definitions expressed with Simple Logic Markup Language (SLML). The logic definitions created by experts need to be broadcast to sub systems and employee terminals. Because the company and departments sub system are typical enterprise system, we use Web Services as the integration approach. Logic definitions are broadcasted to sub systems with SLML through Web Services end points. Going to employee terminals world, there are many types of employee terminals, so we use SLML for powerful terminals that can install Web Services and use WitStream (WS), a text based logic format, for small deices as smart phones and PDAs. All functional components of management console use Witmate engine through JSR94 standard interface. To implement management console, we need a Witmate engine core with Jsr94 cover, SL compiler, SLML parser and WitStream writer.

Company and departments application sub systems are connected with console by Web services through Intranet or VPN. They need to receive plans by SLML, so a Witmate engine core with Jsr94 cover, SLML parser is required for functional implementation. The repository of plans of sub systems is Relation databases, and save plans as WitStream format, so WS reader and writer are required to save and retrieve plans.

Terminals have tow types of structures. For powerful terminals, as PC or UNIX workstations, the Witmate components used are same as company/department sub systems, but plans are saved into files with WitStream format. For thin terminals, as PDA or smart phone, we just need a basic system of Witmate, a Witmate engine core with WS reader, to implement terminals tools, and save plans into memory of devices with WitStream.

This is the system implementation with Witmate logic engine:



Let's take a look at how the plans are defined with SL. As example, This is simple logics of a department expense plan:

```
LogicSet Check_sum
   Def real sum_limit, sum
   #Variables below are input variable.
   Def real travel_cost, over_sea_calls_cost, books_cost
   Let sum_limit=10000 #May be changed year to year.
   Let sum= travel_cost + over_sea_calls_cost + books_cost
   If sum > sum_limit then return false, "Sum over limit"
   If sum <= 0 then return false, "Sum is less than 0"
   Else Fire Check_details(sum, travel_cost, over_sea_calls_cost, books_cost)

LogicSet Check_details
   Def real sum, travel_cost, over_sea_calls_cost, books_cost
   If over_sea_calls_cost/sum > 0.3 And travel_cost > 3000
   Then Return false, "Broke over sea call and travel balance rule"
   If books_cost > 1000 Then Return false, "Bought too much books"
   Else Do apply(sum); Return true, "Application done"
```

Even without more comments, logics of these SL are clear. Witmate will be called from the logic set "Check_sum" with input variables, travel_cost/over_sea_calls_cost/ books_cost, and if sum is OK, "Check_details" logic set will be booted. The Result is a Boolean value to figure out whether process successfully, and a reason message about what is wrong. The reason we use tow logic sets is that the sum and details logics may become more complex year by year, and are managed by two standalone expert teams to be changed separately at different time.

Because of JSR94 compatible, and Witmate core engine exposes a JSR94 like API in j2me platform, the code to load logics into Witmate, and boot logic match is same as any JSR-94 sample code. We ignored this code here.

As an example to show how easy to deal with changes, department users wants to know whether an application are sent successfully to company accountants. Then we may change "Check_details" logic set like:

```
LogicSet Check_details
   …
   If not apply(sum) Then return false, "Fail to send application"
   Else return true, "Sent successfully"
```

This is a part of SLML compiled from SL logics as a brief looking.

```
<?xml version="1.0"?>
<slml>
<!--SLML Generated by SLCompiler[20051110105913] -->
<!-- LogicSet -->
    <lset id="Check_sum">
        <let n="sum_limit" t="dbl">
            <con t="int" s="1">10000</con>
        </let>
        <let n="sum" t="dbl">
            <atom opt="witmate.opt.Math$AddDouble" s="1">
                <var n="travel_cost" t="dbl" s="1" />
                <var n="over_sea_calls_cost" t="dbl" s="2" />
                <var n="books_cost" t="dbl" s="3" />
            </atom>
        </let>
        <lgc>
            <if>
                <and>
                    <atom opt="witmate.opt.Math$GreaterDouble" s="1">
                        <var n="sum" t="dbl" s="1" />
                        <var n="sum_limit" t="dbl" s="2" />
                    </atom>
                </and>
            </if>
            <then>
                <rtn v="1">
                    <con t="bool" s="1">false</con>
                    <con t="str" s="2">
                        <![CDATA[Sum over limit]]>
                    </con>
                </rtn>
            </then>
        </lgc>
        <lgc>
            <if>
                <and>
                    <atom opt="witmate.opt.Math$LessEqualsDouble" s="1">
                        <var n="sum" t="dbl" s="1" />
                        <con t="int" s="2">0</con>
                    </atom>
                </and>
            </if>
            <then>
                <rtn v="1">
                    <con t="bool" s="1">false</con>
                    <con t="str" s="2">
                        <![CDATA[Sum is less than 0]]>
                    </con>
                </rtn>
            </then>
        </lgc>
        <else>
            <fire lset="Check_details">
                <var t="dbl" n="sum" s="1" />
                <var t="dbl" n="travel_cost" s="2" />
                <var t="dbl" n="over_sea_calls_cost" s="3" />
                <var t="dbl" n="books_cost" s="4" />
            </fire>
        </else>
    </lset>
…
```

This is the major part of Logic Engine based implementation approach.

Conclusion

Through this case study, we know the common progress to design, implement a system with Logic Engine based approach. The basic idea of Logic Engine based design and implementation is that let the best people to do the best woks. Developers are responsible for system architecture, data repository and input/output process of Logic Engine, definitions of the system logics are released to experts.

Leveraging the advantaged features of Witmate, we successfully developed and deployed a Logic engine based system into a multiple devices type and multiple networks environment.

We also show a simple example about how easy to deal with changes without any recoding by Logic Engine approach.

References

[1] Witmate users manual: http://www.witmate.com/document.html

[2] JSR-94: http://www.jcp.org/aboutJava/communityprocess/review/jsr094/