**Cloudify Legacy Systems with Architecture Transformation**

# 1. Foreword

As the cloud platform becomes more and more mature, many enterprises consider migrating their legacy systems onto cloud for shorting IT investment and improving the system quality. Migrating the systems manually would bring high cost，as well as high risky and long periods for production. We propose an automatic architecture transformation solution to migrate legacy systems onto cloud platform. In this solution, a domain-specific language is defined, and a library is implemented for transformation implementation. The solution is flexible, language and platform independent.

# 2. The Challenges for Architecture Transformation

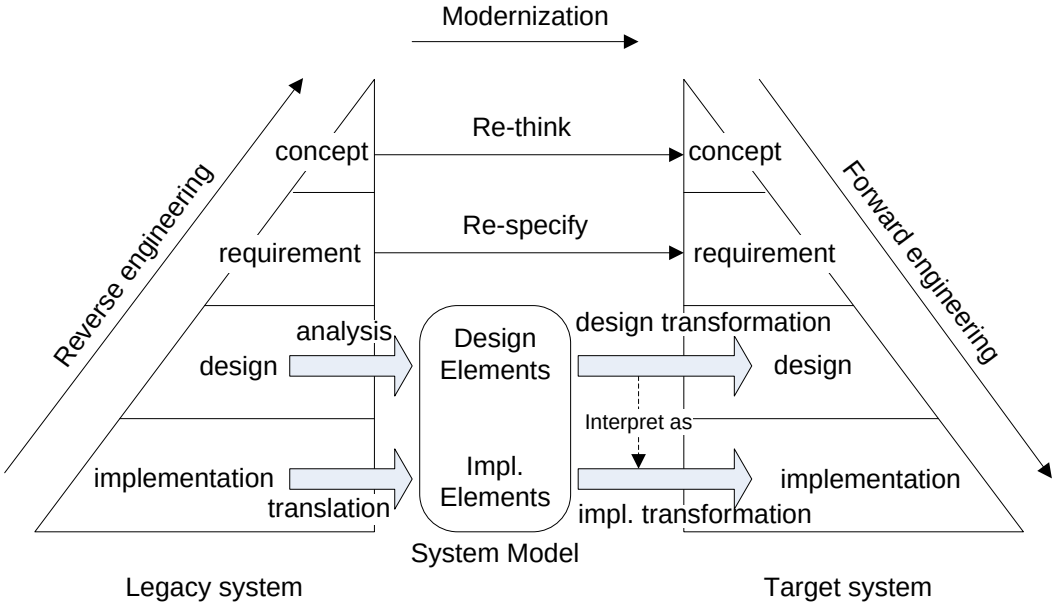


**Fig .1 Architecture Transformations**

Software reengineering using classical horseshoe model consumes plenty of human cost on comprehending legacy systems and rebuild semantic representations

of legacy system. Transform the legacy systems to target platform at design, even implementation level would improve the efficiency, as shown in Fig.1. The architecture transformation faces some critical problems.

1. How to locate the source code to transform. For the same business logic, different developers may bring different design and implementation styles.
2. Architecture transformation is context-sensitive. The context should be clear when trying to transform a design, which cannot be solved by only syntax analysis.

# 3. Transformation Solution

Cloud platform becomes more and more popular in enterprise applications. Many corporations are trying to migrate systems onto cloud. We propose a migration solution towards cloud with automatic architecture transformation, as in Fig.2.

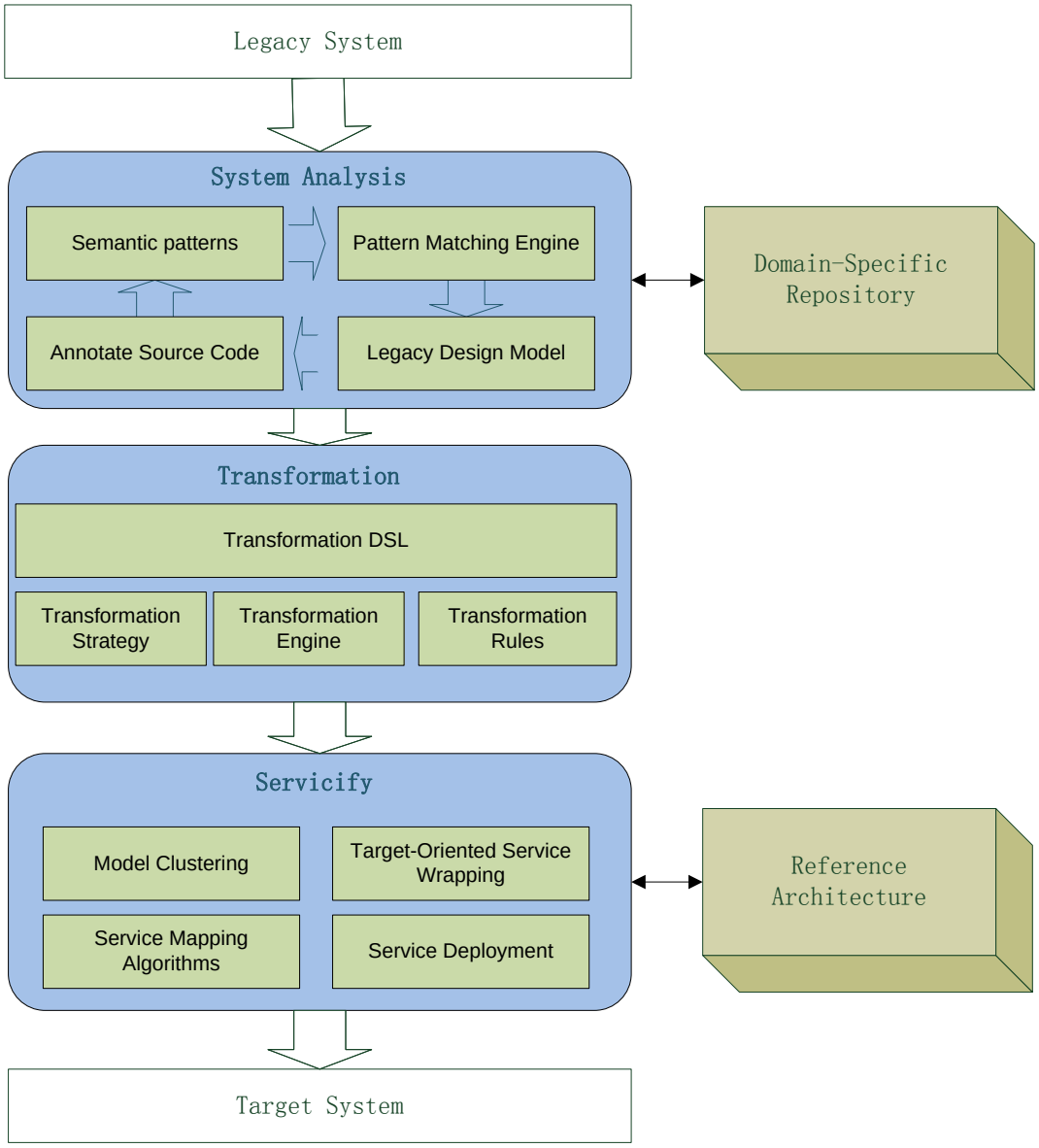


**Fig .2 Automatic architecture transformation process**

1. **Annotating source code using pattern matching**. Semantic terms are instrumented into the source code from the source code and reference architecture views. The annotations are represented as comments at different levels, e.g. classes, methods or expressions.

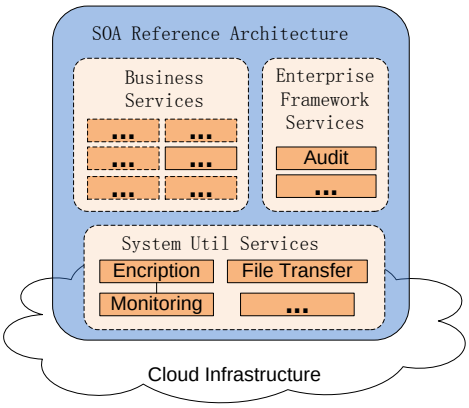


**Fig 3 SOA Reference Architecture**

a) Semantic patterns are extracted from reference architecture. Take SOA as an architecture example, the services in SOA can be grouped into business domain services, enterprise framework services and system utility services, as shown in Fig 3. Semantic patterns can be defined for each group of services.

b) Pattern matching techniques are adopted for mapping semantic patterns to source code. Some code elements would be identified for software architecture, business rules, domain objects and functions, etc.

c) The identified elements will help program comprehension, and add more annotations to source code.

d) As the understanding of source code, more semantic patterns can be obtained and used for further analysis. The code can be instrumented iteratively.

2. **DSL-Based Architecture Transformation**. A Ruby-based domain specific language is defined for transforming architecture A to architecture B. The transformation process is shown in Fig.4. In this process, java source code is parsed using ANTLR into AST. Architecture is transformed on AST using DSL. Source patterns define the source code styles for transformation and the transformation profile lists the transformation rules for transforming a source pattern to what target code. StringTemplete is for transforming modified AST to Java code.

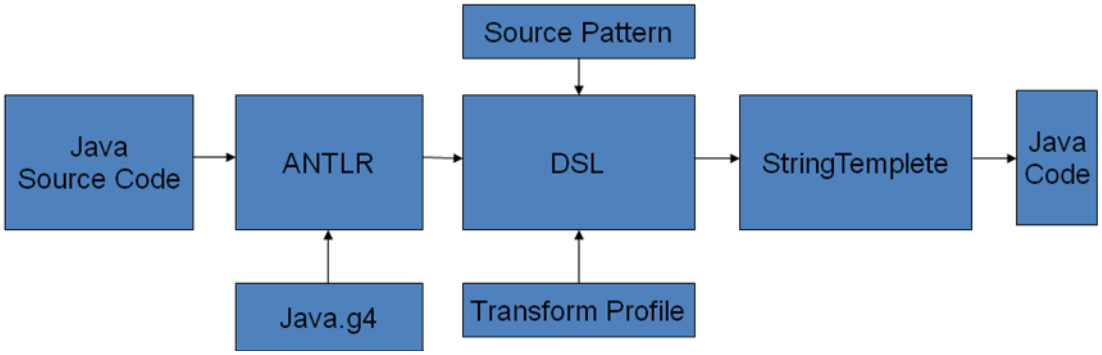


**Fig 4 Transformation Process for Java Code**

a) A ruby DSL is defined for architecture transformation. We defined the DSL syntax for engineering. When transforming java source code, write the transformation logics using this syntax.

i. First, locate the source code to transform, as shown below. The command returns the source code fitting the source pattern.

```
discover "src" do
    match "source pattern"
end
```

ii. The discovered source code would be recorded or represented to engineers for analysis, so *print* action is defined.

```
print "src" do
    to_file "fileName"
    to_console
end
```

iii. Automatic transformation rules can be written using the *replace* command. In this command, *to* clause shows the target pattern and *todo* marks what cannot be transformed automatically.

```
replace "src" do
    if condition
        to "target pattern A"
    else
        to "target pattern B"
    todo "auto failed, need manual transformation"
end
```

iv. We take java file operations as examples, as given in Fig.5-Fig.7.

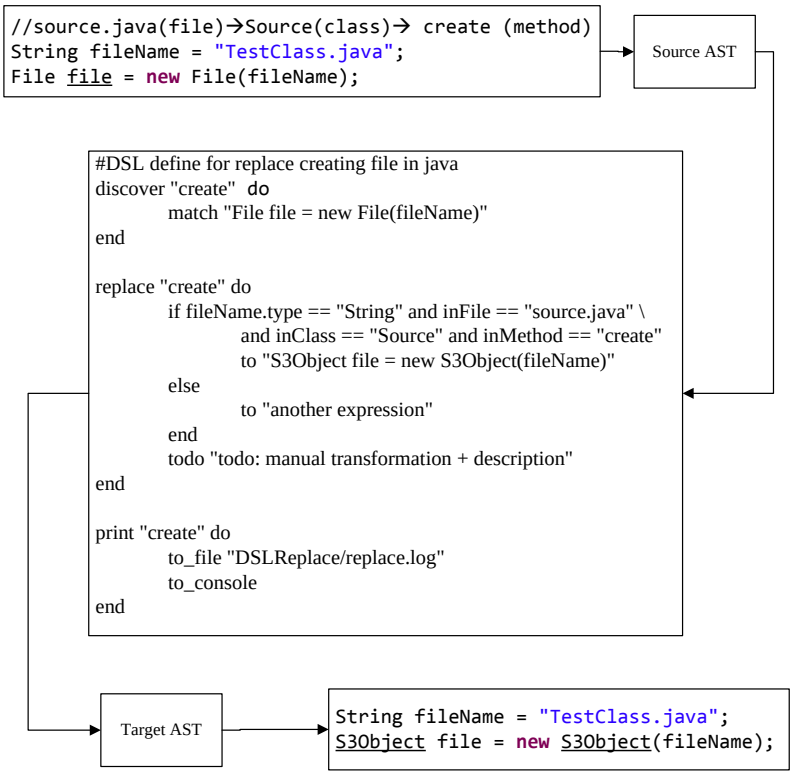


Figure 5 the replace process of creating file


Figure 6 the replace process of reading file


Figure 7 the replace process of copying file

b) When implementing the DSL, some other techniques can also be used, including model clustering with dependency analysis, transformation engine and transformation rules.

i. Transformation Engine. A general engine is the fundamental of architecture transformation. In the engine, a unified language, like AST, is used to represent the source code.

ii. Transformation Rules. The rules are pre-defined for architecture transformation.

iii. Transformation Strategy. The architecture is transformed in which strategy.

3. **Servicify**. The transformation brings an intermediate system having the same business logics with original system. However, it usually does not take sufficient advantages of target architecture. Thus, some redesign is essential.

a) For SOA reference architecture, the service is extracted from the legacy system. The extraction needs clustering source code that is dependent by the concerned code.
b) Target-Oriented service wrapping. Different target architecture or platform may requirement different services. So it is necessary to provide target architecture description for specific transformation.
c) Service mapping is an important technique for merging services from different applications. For services with different implementation, even different languages.
d) Generate target services and deploy them onto the target platform.

In a word, the transformation process needs lots of transformation techniques and tools. We are focusing on the transformation DSL design, implementation, and source annotations. Some existing open source tools will also be involved into this process to build a complete transformation toolkit.

# 4. Highlights of the Solution

The transformation solution with reference architecture has following highlights:

1. Flexible DSL. Transformation DSL is defined and implemented using Ruby. The DSL is readable for engineers and extensible for designers.
2. Language and platform independent. In the transformation engine, a middle language – Abstract Syntax Tree – is used for transformation. The transformation from source to target can be implemented in AST mode.
3. Source patterns are defined from both semantic and architectural views. Code annotations are instrumented to improve the code semantics for better comprehension.
4. SOA is used as default reference architecture. Service-oriented refactoring enhances the system reusability. It is helpful for shorting IT investment.