Optimization of Software Production using Generative Programming

Senthil Jayavel, Arumugam S, Brijendra Singh, Sanklan Saxena

School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, Tamil Nadu 632014, India.

ABSTRACT

Background: Optimization in Software Production implies to minimize the time and resource consumed in production which can be achieved by minimizing the code length during implementation, achieving reduction in cost and time of production. Objective: To construct a Generative Paradigm based System to obtain optimization in Software Production. Results: We suggests a Theoretical Model System(T) works on Generative Paradigm which input a set of Software under production (say S), which are from same domain and having approximately same number of components and Database Tables with their corresponding sets of UML, class diagram, package diagram, ERD and Activity diagrams. System T generates a domain common -package diagrams, class diagrams, Activity Diagrams with UMLs which simplifies the design of S. Then System T generates package diagrams and class diagrams, Activity Diagrams with UMLs for every element of set S using domain common production. Conclusion: Code reusability is highly enhanced and length of code to be written is considerably declined, leading to time saving and reducing investment in Software Production.

INTRODUCTION

Optimizing Software Production by decreasing resources required in implementation of Software systems (belonging to same domain) under production through generalizing of the code part. API's or other reusable codes are by-products generated during production of particular software, which are used in future production, thus giving reusability of code written. Since the API's are not designed for specific domain, it creates generalization problem (Frank Pfenning and Yannis Smaragdakis, 2003) production of Tangled Code or due over generalization, introduction of unusual variation point, creating ad-hoc generalization. In this paper, generalization Problem in Object Oriented Paradigm is solved by implementing a Generative Paradigm based System which takes input from output of the Design and Analysis phase of software production and outputs the input of the implementation phase. The theoretical System generates a domain common task to be done which enhance the generalization and enhance the reusability. Since the APIs are domain specific, in domain software production, the ease of use is highly enhanced.

The very first part of the paper introduces Generative System, its inputs and outputs. The next part is algorithm on which system can work. Then paper analyzes the system and optimization obtained. Then it discusses how enhancement of code reusability is done and the concluding portion discusses the challenges we have in implementing the system and what is the future of the technology introduced.

Generative System:

Input:
A Set of Software (say S) going under same production phase (Iqbal,M. and Rizwan,M.,2009;Hamou-Lhadj et al.,2009) belongs to same domain (Domain here- implies towards same production field; for example billing and booking websites). Assuming that number of components (Software Systems are divided into independent components (Diomidis Spinellis, 2001; Krzysztof Czarnecki et al., 2002. A component is smallest independent unit of system which is meant to do a given task with defined input and expected output.) in each system s,E S is approximately n with U i as set of UML Models belong to s i system, with Cl i class diagram set, P i package diagram set and ERi ERD set and ADi Activity Diagram set correspondingly where s i is a software system under production and i is whole-number.
Output:
The theoretical system generates a U c UML Model for domain S with Cl c class diagram set, P c package
diagram, ERc ERD set and ADc Activity Diagram set. Using this domain common architecture, a new optimized
OUi UML diagram set with OCl iclass diagram set, OP i package diagram set and OER i ERD set and OAD i
Activity Diagram set is generated by the system for all si.

Method:
N: Total number of system under production in Domain S; where N ∈ Z+.
n: (assumption) number of component systems in every system si.
Thus total number of component system we have in our Domain is N*n.
Ci is set of component system belongs to each system si of Domain S. For each system si, each component is
denoted cij designed such that the dependencies between them two component is zero or minimum. Thus other
architectural design also divided accordingly.
UMLs of each system s i has components UMLs, we got u ij corresponding to every component cij where uij
belongs to U i , which is of system si. Thus updating class diagram set, package diagram set, Activity diagram
set and ERD set components of each system si corresponding to each component cij and UMLs.
Consider Set CCi (common component) where (1<i< N*n+1) stands for number of component integrated to
get common component cc ij with common alities(commonality here defines task similarities) where j is a
positive integer. Thus ccij defines a member of CCi which is common component of i components of Domain S.

Integration – Assume that we have selected i components from all components present in Domain S such
that all i component have approximately common task to perform and a general implementation can be done for
them, which can be reused and hence that general implementation is known as common component cij obtained
from integration of i components and j th selection. Integration can be done by finding out commonalities
between components of different system which can be done by comparing task similarities, inputs and outputs,
ERD, Activity diagram or other diagrams to components of different systems.

Degree of Generalization (DoG(S)) is defined as to scale the reusability introduced in S.
Vote Level of a common component c ij is an integer parameter which is equal to number of component in
Domain S is integrated to obtain that common component. Vote Level of ccij is mathematically equal to i.
VoteLevel(Ai): is number of common components at i th vote level provided i>1.
Since, in a system; two components cannot have task similarity as different component of same system
cannot have same objective/task to do(even if two components in a system has same task to do, the component
won’t be implemented twice instead same implementation can be used twice). Hence to get cc ij, analytical
system select i components from different systems in S.

Number of Common Component at level of integration i (CC i) is less or equal to NC i where (1<i<N).
Thus, ∑ A i*i ≤ N*n for (1 <i< N), A i ≥ 0 and A i, i are integers.

Calculating Degree of Generalization and throughput of Domain S:.
C[i][j][j]: j th component c of i th system in Domain S include[i][j][j]: Boolean to describe inclusion of cij in any
common component i.e. is false if cij is not common with any component in Domain S. CC(i,j,list)-following description j:component number list: containing system number and corresponding
component number of all components common to CC.
i-the vote level of common component i.e. number of components in list.
counter: counter holds the component number of current common component initialized to one.
Ai: number of common components with vote Level i.

For i=1 to N
for j=1 to n
if include[i][j]=false
CC(i , j , list)=( 1, counter++ , c[i][j] )
for k: (i+1) to N
for l: 1 to n
if c[i][j] has commonalities’ with c[k][l]
add c[k][l] to list of CC( i, j )
增加 vote level of CC i.e. CC(i++, j)
For i=2 to N
if CC(i)=i
Ai=Ai+1
To calculate Degree of Generalization: DoG(S) = (∑ A i ) / (∑(A i * i))
To calculate the throughput = \[
\frac{\text{number component to be implement after optimization}}{\text{number component to be implement before optimization}}
\]

\[
\frac{N \times n - \left( \sum_{i} A(i) \times i - \sum_{i} A(i) \right)}{N \times n}
\]

Generating Domain Common Design Architect like UML class diagram set, P class package diagram, ER ERD set and AD Activity Diagram set from all selected for generalization in above step. Implementing Above, generating APIs that are of high usability for all Systems under production in Domain S. Using Domain common APIs, System generates a new optimized OU UML diagram set with OCl class diagram set, OP package diagram set and OER ERD set and OAD Activity Diagram set for all systems under production in Domain S.

![Diagram](image_url)

**Fig. 1:** Obtaining common component by integration of k components

**Results:**
Assume that N systems are in production with approximately n component system in each system. Since, all system belongs to same domain if the degree of resemblance (at domain level) to be 5 percent; it implies that 5 percent component systems are approximately have same functionality or roles or job or task. Thus 5 percent of total Components are common with Vote Level ‘N’.

Total number of components in Domain = N*n

Total number of common components selected for Domain Common Architect = \( \sum A_i \)

Since Vote Level of each Common component is assumed as i where \( N > i > 1 \), \( i \in \mathbb{Z} \)

Number of component debar from production = \( \sum A_i \times i - \sum A_i \) where \( N > i > 1 \), \( i \in \mathbb{Z} \)
Let’s assume that N = 1000 and n=100 with making an assumption that S get 50 cc i1 where i = {76, 77,… 100} which implies that domain get one common component at every vote level which is greater than 75 provided every component is of same cost. Calculating DoG(S)= 0.011363636

Number of Component saved from implementation = (76*1+77*1… 100*1)-(50) = 2200-25=2175.

Throughput = (97825/100000) = 0.97825 which implies that System has saved 2.175% (approx) of total resources.

Let’s assume that N = 1000 and n=100 with making an assumption that S get 50 cc i1 where i = {51, 52,… 100} which implies that domain get one common component at every vote level which is greater than 50 provided every component is of same cost.

Calculating DoG(S) = \[
\frac{50}{51*1+52*1+\ldots+100*1} = 0.01324
\]

Number of Component saved from implementation = (51*1+52*1… 100*1) -(50) = 3775-50=3725.

Throughput = (96275/100000) = 0.96275 which implies that System has saved 3.7225% (approx) of total resources.

Since, it is not possible that all components have same cost. A survey was done among students who were doing mini-project, the domain S was drawing Softs having projects: Photo editor, Paint, Shared Drawing etc. Every student was asked to divide their respective project into approximately 11 components and give each component a resource consuming number (hours to design, implement and validate) which defines how much resource the component has consumed. One sample common components of the domain S is given here. The analysis after applying the above theorem to survey.

Tool state manager
- Holds reference to every tool in every application.
- Stores default state of every tool.
- Stores state set of every tool.
- State of a tool defines set of every property with valid value on it that can be defined on a tool.
- Method to set default state or user defined state on reference tool.
- Method to get state of a reference tool.

Properties of Common component:
The common component shown here is highly generalized and dynamic. Since, the Tool State Manager doesn’t know which tool it is referencing and that the state of tool referenced will be dynamically defined on the basis the tool which is referenced. Thus, it gives a high level abstraction to design and further because of this abstraction common component have high reusability.

<table>
<thead>
<tr>
<th>Table 1: Domain S</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Si</td>
</tr>
<tr>
<td>C1</td>
</tr>
<tr>
<td>C2</td>
</tr>
<tr>
<td>C3</td>
</tr>
<tr>
<td>C4</td>
</tr>
<tr>
<td>C5</td>
</tr>
<tr>
<td>C6</td>
</tr>
<tr>
<td>C7</td>
</tr>
<tr>
<td>C8</td>
</tr>
<tr>
<td>C9</td>
</tr>
<tr>
<td>C10</td>
</tr>
<tr>
<td>C11</td>
</tr>
</tbody>
</table>
Table 2: Domain Cost before optimization

<table>
<thead>
<tr>
<th>System</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>221</td>
</tr>
<tr>
<td>2</td>
<td>217</td>
</tr>
<tr>
<td>3</td>
<td>350</td>
</tr>
<tr>
<td>4</td>
<td>312</td>
</tr>
<tr>
<td>5</td>
<td>408</td>
</tr>
<tr>
<td>6</td>
<td>364</td>
</tr>
<tr>
<td>7</td>
<td>277</td>
</tr>
<tr>
<td>8</td>
<td>378</td>
</tr>
<tr>
<td>9</td>
<td>346</td>
</tr>
<tr>
<td>10</td>
<td>212</td>
</tr>
<tr>
<td>11</td>
<td>240</td>
</tr>
<tr>
<td>12</td>
<td>274</td>
</tr>
<tr>
<td>13</td>
<td>378</td>
</tr>
<tr>
<td>14</td>
<td>353</td>
</tr>
<tr>
<td>15</td>
<td>354</td>
</tr>
<tr>
<td>16</td>
<td>284</td>
</tr>
<tr>
<td>17</td>
<td>265</td>
</tr>
<tr>
<td>18</td>
<td>315</td>
</tr>
<tr>
<td>19</td>
<td>274</td>
</tr>
<tr>
<td>20</td>
<td>382</td>
</tr>
<tr>
<td>Domain Cost</td>
<td>6204</td>
</tr>
</tbody>
</table>

Table 3: Integration Table

<table>
<thead>
<tr>
<th>Common Component $cc_{ij}$</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>$cc_{1,1}$</td>
<td>$cc_{1,1}, cc_{1,2}$</td>
</tr>
<tr>
<td>$cc_{1,2}$</td>
<td>$cc_{1,2}, cc_{1,3}$</td>
</tr>
<tr>
<td>$cc_{1,3}$</td>
<td>$cc_{1,3}, cc_{1,4}$</td>
</tr>
<tr>
<td>$cc_{2,1}$</td>
<td>$cc_{2,1}, cc_{2,2}$</td>
</tr>
<tr>
<td>$cc_{2,2}$</td>
<td>$cc_{2,2}, cc_{2,3}$</td>
</tr>
<tr>
<td>$cc_{2,3}$</td>
<td>$cc_{2,3}, cc_{2,4}$</td>
</tr>
<tr>
<td>$cc_{3,1}$</td>
<td>$cc_{3,1}, cc_{3,2}$</td>
</tr>
<tr>
<td>$cc_{3,2}$</td>
<td>$cc_{3,2}, cc_{3,3}$</td>
</tr>
<tr>
<td>$cc_{3,3}$</td>
<td>$cc_{3,3}, cc_{3,4}$</td>
</tr>
</tbody>
</table>

Cost after optimization = (total cost) + (cost of $cc_{ij}$) – (cost of deprived components) = 6204+45+32+12-(45*6+7*32+12*10) = 5679 throughput of Domain $S = 5679/6204=0.915377$

cost saved=6204-5679=525 $DoG(S) = 89/ (45*6+7*32+12*10) =0.144$

Discussion:
Thus, if Domain is selected intelligently or in other words if software systems under production are distributed intelligently, the degree of resemblance and Vote Level of corresponding common component will high, the reusability will highly enhanced. Another important result is lower the $DoG(S)$, higher re-usability.

Conclusion:
From analysis part, it is be concluded that “the larger the domain, the greater the optimization” and secondly “greater the degree of resemblance, greater optimization”. The second point implies that greater concern should be given while creating domain. Since, above production technique generates Domain components which can be reused in any further production in that domain as the components are domain specific.

One of the greatest challenge that the above technology is to find the intelligence in Theoretical system to integrate the component with other component and find resemblance, so that system can be automated since in survey and project analysis the common component was find out manually. How we are achieving reusability? : Let assume, a Common component of Vote Level N, Which implies that this component is required N times in the Domain S. So if component is implemented, it can be reused N-1 times.

REFERENCES