A Constructive Approach To Software Evolution

Selim Ciraci, Pim van den Broek, Mehmet Aksit
Software Engineering Group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente
PO Box 217
7500 AE Enschede
The Netherlands
Email: {ciracis, pimvdb, aksit}@ewi.utwente.nl

***Abstract*— In many software design and evaluation techniques, either the software evolution problem is not systematically elaborated, or only the impact of evolution is considered. Thus, most of the time software is changed by editing the components of the software system, i.e. breaking down the software system. The software engineering discipline provides many mechanisms that allow evolution without breaking down the system; however, the contexts where these mechanisms are applicable are not taken into account. Furthermore, the software design and evaluation techniques do not support identifying these contexts. In this paper, we provide a taxonomy of software evolution that can be used to identify the context of the evolution problem. The identified contexts are used to retrieve, from the software engineering discipline, the mechanisms, which can evolve the software software without breaking it down. To build such a taxonomy, we build a model for software evolution and use this model to identify the factors that effect the selection of software evolution mechanisms.**

Keywords: Software Evolution, Software Architecture Synthesis, Software Evolution Taxonomy, Software Evolution Framework.

I. Introduction

Due to demand from users and changes in environment and organization [1] software systems need to evolve. Due to this, the initial requirements of the system are changed. One type of change is the addition of new requirements to the system. Thus, software evolution for such changes involves finding solutions for these new set of requirements and integrating them into the system without effecting the quality of the system. We call this the *integration* problem.

In the literature, as we detail in section II, the evolution problem is not systematically worked out in problem solving based techniques (e.g. Synbad [2]) or only the impact of the changes is calculated using scenario-based techniques [3]. That is, the mechanisms that can ease software evolution are not considered. For example, for a given change scenario, the context of this change can be identified and the most applicable techniques that reduce the impact of change can be selected. However, these steps are not included in any evaluation technique. Obviously, there are many mechanisms in the software engineering domain that can be used to evolve software. Even the inheritance mechanisms provided by object-oriented languages can be used to cope with some evolution requests. However, the contexts where these mechanisms are most applicable is not identified. So, there is a gap between the software design and analysis techniques and solution mechanisms (such as styles and patterns). To close this gap, we need a mapping mechanism in which the contexts of the evolution problem in consideration are identified and these contexts are used to find the set of mechanisms that are applicable.

In this paper, our aim is to provide such a mapping between software design/analysis techniques and design patterns/styles (which we call mechanisms) for the *integration problem*. We propose to add steps to the design process, in which:

1) After solutions to the initial requirements are found, the solutions that are expected to change are identified (using evaluation techniques like scenarios), the contexts of these evolution problems are found and these solutions are extended with the mechanisms that provide an extensible interface to this evolution problem.
2) The solutions to changed requirements are found, the contexts of these evolution problems are identified and using these contexts the mechanisms that allow composition of old solutions with the new ones are extracted.

To achieve this aim, we first identify the types of changes that occur in requirements due to evolution and formulate the constructive model based on these changes. Then we focus on *integration* problem and we use the constructive model to identify the contexts of these problems. In other words, we identify the factors that affect the selection of the mechanisms from software engineering domain. Then, we provide a framework of solutions that are applicable to each context.

This paper is organized as follows: In the next section we provide an overview of software design and architecture evaluation techniques and identify their problems with respect to software evolution. The software evolution model is described in section III. We present the taxonomy of software evolution in section IV. For all identified contexts, we list mechanisms that can be used to cope with evolution in section V. We

This work has been carried out as a part of the DARWIN project under the responsibilities of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

conclude and provide the future work in section VII.

II. Software Evolution in Software Design and Software Evaluation Techniques

In this section, we describe what we mean by the *gap* between software design/evaluation techniques and design patterns/styles. We consider the most well-known design and evaluation techniques and describe how identifying the context of the evolution problem helps in the choice of the software evolution mechanisms.

A. The Unified Process

The Unified Process [4] is a use-case driven, iterative and architecture centric software design process. The life of a system is composed of cycles and each cycle concludes with a product. Each cycle is divided into four phases. The first phase is the inception phase and in this phase the requirements are analyzed and a general vision about the product is developed. This phase is followed by the elaboration phase in which the architectural baseline of the product is developed. During the third phase the product is built and this phase is labelled as construction. The last step, called transition, involves the manufacturing of the product.

To support evolution in Unified Process, there must be link between the transition phase of the previous cycle and the inception and elaboration phases of the current cycle. With this link, the designer, while gaining a perspective about the old system, can also develop ideas about integrating new requirements to the system. That is, with this link the designer can identify the evolution problem he is faced with, select the suitable evolution technique and then apply this technique to the design. For example, if the new requirements extend the current system, the designer can choose to delegate the current system with new requirements. Thus, the new system can be designed using means of delegation mechanisms like call forward protocols.

B. Software Architecture Synthesis Process

The Software Architecture and Synthesis process (Synbad) [2] is an analysis and a synthesis process, which is a widely used process in problem solving in many different engineering disciplines. The process includes explicit steps that involve searching solutions for technical problems in solution domains. These domains contain solutions to previously solved, well established, similar problems. Selection of which solution to use from the solution domain is done by evaluating each solution according to quality criteria.

The method consists of two parts, which are solution definition and solution control. The solution definition part involves identification and definition of solutions. In this part client requirements are first translated into a technical problems; these are the problems that are actually going to be solved. These technical problems are then prioritized and a technical problem is selected according to this priority order. The solution process involves identifying the solution domain for the problem and searching possible solution abstractions in this domain. Selected solution abstractions are, then, extracted from the solution domain and specified to solve the problem in consideration. In the last step of the solution definition part, the specified solutions are composed to form the architectural description of the software. The solution abstractions may cause new problems to be found; thus there is a relation, labeled as 'discover', between solution abstraction and technical problem.

The solution control part of Synbad represents the evaluation of the solutions. The evaluation conditions (e.g. constraints on applying the solution) are provided by the subproblem and by the solution domain. The solutions extracted from solution domains are expressed as formal models for evaluation. Then optimizations are applied to the formal model in order to meet the constraints and the quality criteria. The output of these optimizations is then used to refine the solution.

Synbad treats each problem separately and the solutions of each problem are composed to form the solution of the overall problem the software is going to solve. Thus, this process inherently supports the addition of new requirements to evolve the software. When new requirements arrive, their technical problems are analyzed and the solution abstractions for these technical problems are extracted from the solution domain. Each extracted solution abstraction causes a new technical problem to be identified, which can be stated as "given a solution, what are the techniques to compose this solution to the system". For this problem, the solution abstraction and the system define the quality criteria and constraints. Here, the quality criteria are the non-functional requirements of the system. The constraints, on the other hand, are the factors that affect the selection of the composition mechanisms. For example, if the extracted solution is already implemented and its source code can not be changed, then the composition mechanism should be a run-time solution. In this paper, we provide a taxonomy that lists all these constraints. Thus, the software engineer can identify the evolution problem he is faced with and search for the mechanisms accordingly.

C. Scenario-based Evaluation Techniques

There are many scenario-based techniques that evaluate software architectures with respect to certain quality attributes [3]. Scenario-based Architecture Analysis Method (SAAM), for example, is a method for understanding the properties of a system's architecture other then its functional requirements [5]. The inputs to SAAM are the requirements, the problem description and the architecture description of a system. The first step of SAAM is scenario creation and software architecture description. During this, all stakeholders of the system must be present; scenarios are considered to be complete when a new scenario doesn't affect the architecture. In the last step, scenarios are evaluated by determining the components and component connections that need to be modified in order to fulfill the scenario. Then the cost of modifications for each scenario is estimated in order to give an overall cost estimate.

In recent years, SAAM has been specialized to focus on a quality attribute like modifiability [6] and extended to find the

trade-off between several quality attributes [7]. These methods can easily be used or adapted to find the impact of evolution requests. Though, after finding the impact, software engineers are faced with the problem of finding the mechanisms that are applicable to the evolution problem in consideration. When with scenarios certain components are found to be hard to evolve, how can the software engineer make them easier to evolve? For this, the evolution problem should be analyzed in detail; the constraints of the software system and the evolution mechanisms should be identified and the most applicable mechanisms should be used to replace/change the components. That is, the context of the software evolution should be identified in order to select the applicable mechanisms. Currently, none of the evaluation techniques has steps that include such analysis. In this paper, we identify the contexts of evolution problems and mechanisms. Thus, after finding the impact, the software engineer can find the applicable evolution mechanisms by selecting the context of the problem he is dealing with. Furthermore, in this paper, we also list some mechanisms that can work in the identified contexts.

D. Design Pattern and Styles

In the software engineering domain there are many mechanisms that can cope with various evolution problems. Some design patterns, for example, make it is easier to add new behavior to the system. In their study of comparing design patterns to simpler solution for maintenance, Prechelt et al. [8] concludes that due to new requirements design patterns should be used, unless there is an important reason to choose the simpler solution, because of the flexibility they provide. Mens and Eden [9] list some of these evolution mechanisms and determine how helpful they are for some evolution situations. Their analysis shows that these mechanisms are very costly to use for certain evolution problems while for others they are not. This shows that there are contexts for these techniques. Thus, identifying these contexts and then selecting the mechanism to use may greatly ease the procedure for the evolution of software.

The problem here is that these contexts are not analyzed. We know that design patterns and styles can ease evolution operations but what the applicable mechanisms are for a given evolution problem is not known.

III. The Model of the Constructive Approach to Software Evolution

In this section, we formulate the technical problem and a model for software evolution. There are many studies that try to capture the scope of evolution. For example, Bennett and Rajlich [10] state that software evolution occurs only after the initial software system is developed. We consider evolution as a procedure for adding the set of changed requirements to the software system. Thus, evolution does not only occur after the initial system is developed, since user requirements may also change during the development of the initial system.

A software system starts its life cycle with a set of client requirement specifications denoted by R_{System}. By using some design process, the solutions for these requirements are found as presented in Figure 1. Here, a solution is a set whose elements are software components, such as classes, methods, attributes, relationship between classes, and implementations of methods (e.g. a set with two classes and an inheritance relation between them), and is denoted by S. The elements of a solution set depend on the design process used. For example, if Unified Process [4] is used as the design process then the solutions are classes and interactions between classes. These solution sets are the elements of the set of solutions to the system S_{System}.

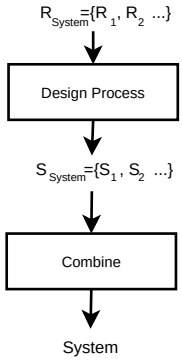

Fig. 1. The Software Design Process

In order to find a solution for the overall problem that software system is going the solve, the solutions in S_{System} should be combined; that is the interactions between the solutions should be identified. Thus, we introduce the $Combine$ operator which refers to the process of composing the solutions:

$$System = Combine(S_{System})$$

Evolution causes changes in the requirements; that is the elements of the set R_{System} are changed. Using this, we identify three types of changes:

- Integration: Refers to the type of change where the solution, S_{New}, of a new requirement is to be added to system. $S_{System} \Rightarrow S_{System} \bigcup \{S_{New}\}$.
- Removal: Refers to change where a requirement is removed from R_{System}, thus the solution corresponding to this requirement is also to be from the system. $S_{System} \Rightarrow S_{System} - \{S_{Old}\}$.
- Modification: This type captures the changes where a requirement in the set R is modified. Thus, the old solution, S_{Old}, of this requirement is replaced by, S_{New}, the new solution. $S_{System} \Rightarrow (S_{System} - \{S_{Old}\}) \bigcup \{S_{New}\}$

As shown above, the changes in the requirements causes changes in the solutions of the system. Thus, to achieve the new system the combine operation is restarted with this changed solution set. This is the destructive approach to software evolution. Without considering the applicable evolution mechanisms during the design phase, the results of the old combine operation are broken down and the operation is restarted with changed S_{System}. A better approach is to

find mechanisms that allow composition of changed solutions (contained in S_+ and S_-) to the system without breaking down the system. We model this approach as:

$$NewSystem = (S_+, S_-) \bigoplus System$$

In the above definition, the set S_+ contains the solution to be added and S_- contains the solution to be removed from the system; $System$ is the system that has already been built, $NewSystem$ denotes the system that is to be achieved. The $\bigoplus$ operator defines the process of finding the context of the evolution problem and then the applicable mechanisms, which allow evolution without breaking down the system, at that context. The mechanisms to be used greatly depends on the type of change; thus for the identified three types, we define the constructive approach as:

- Integration: $(\{S_{New}\}, \{\}) \bigoplus System$
- Removal: $(\{\}, \{S_{Old}\}) \bigoplus System$
- Modification $(\{S_{New}\}, \{S_{Old}\}) \bigoplus System$

In this paper, we focus on the *integration* and list the mechanisms that allow a constructive approach for this type of change.

In many cases, the software engineers want to design their initial systems so that they can handle some evolution requests. To identify the components that are going to be effected by evolution often scenarios are used. In our model, these scenarios can be used as future requirements and then the software engineer can identify the components that are going to be affected by evolution. Then, using the taxonomy we present in this paper, the software engineer can identify the context of the evolution problem, find applicable solutions to this problem and extend the system with these solutions.

To clarify this model for evolution, we examine the PDA input and storage system example given by Noppen, Van den Broek and Aksit [11]. The requirements of this system are:

- R_1: The system should be able to accept textual input from the user.
- R_2: The system should be able to accept spoken input
- R_3: The system should be able to store the given input in text format on a local disk.

Thus $R_{system} = \{R_1, R_2, R_3\}$. For this example, we use Unified Process as our design procedure and we find the following solutions: $S_1 = \{C_1, C_2, R_1\}$, $S_2 = \{C_3, C_4, R_2, R_3\}$, $S_3 = \{C_5\}$ where (C stands for Component):

- C_1: Abstract I/O Reader class
- C_2: Keyboard Reader Class
- R_1: Inheritance relation between C_1 and C_2
- C_3: Audio Recorder class
- C_4: Voice Recognizer class.
- R_2: Inheritance relation between C_1 and C_3
- R_3: Aggregation relation between C_4 and C_3
- C_5: File writer class.

The overall solution to the PDA Input and Storage System is:

$$System = Combine(\{S_1, S_2, S_3\})$$

Assume that after the initial release, the users of the system demanded that system should be able support encrypted file writing. We solve this requirement by introducing the class *EncryptedFileWriter*. So we have:

$$NewSystem = (\{EncryptedFileWriter\}, \{\}) \bigoplus System$$

Thus we need a mechanism to compose this class with the old system and we show how our taxonomy supports finding this mechanism in the remaining sections of the paper.

IV. Taxonomy of Software Evolution

The software engineering domain contains many mechanisms to the problem of evolution. Obviously, every mechanism has a context where it is applicable. Thus, we need to identify the contexts of the evolution problems and then try to the find the mechanisms; in other words, we need to build a taxonomy of software evolution.

In Section III, we defined the $\bigoplus$ operator in which the context of the evolution problem is identified. For the integration evolution problem, this operator works by finding the contexts for the solution S (we refer to S_+ as S in the reminder of the paper) and extracting the mechanisms applicable for these contexts. So, to find the context of the evolution we need to categorize the relationship between S and S_{System}. We identify three parameters that categorize this relationship. The first parameter (CHAR) defines the characteristic of the solution S; it ranges over {E,C,Ex}, where:

- Extension(E): The demands (e.g. marketing) can show the near future expected changes. Thus, we can *extend* the system so that when these changes happen, they can be easily added to the system. The solutions that are going to be changed or added to the system are identified by means of scenarios. The new solution set, S, contains software components that are affected by the scenario.
- Composition (C): The changes have happened and the solutions for the new requirement are found. Thus, $NewSystem$ is defined by composing $System$ and S. For this value the new solution set, S, contains software components that solve the new requirement and the software components that are affected by this requirement.
- Exception (Ex): No solution to the new requirement can be found. Thus, S does not exist.

The second parameter, denoted by REL, specifies the relationship between the system and the solution in consideration, which is the intersection of the sets S_{System} and S. To identify this relationship, the solutions to the new requirement should exist. This parameter takes values from {NO, O, S, I}, where:

- Non-overlapping(NO): S and all of the solutions in S_{System} do not share software components; that is $\forall S_j \in S_{System}, S_j \cap S = \emptyset$. With the destructive approach, since there is no intersection between solutions, the S is added to the system by the *Combine* operation. In the constructive approach, on the other hand, the *System* is not broken down to its solutions, so the contexts where REL $= NO$ should include mechanisms that bind the new solution to the system.

- Overlapping(O): In this case, S and at least one solution in S_{System} share software components. For example, the addition of the new solution, S, to the system may cause some parts of the old solutions to be replaced by the new ones. This case can be presented with our model as: $\exists S_j \in S_{System}, S_j \cap S \neq \emptyset$. For these extensions, substitution techniques in which the solution, S, replaces or parts of it replace a solution in S_{System} can be used. This case has two special cases:
  - Specialization (S): The new solution extend the system; that is $\exists S_j \in S_{System}, S_j \subset S$. The obvious solution to this evolution problem is building a delegation mechanism.
  - Interpretation (meta-layers)(I): In this case, the new solutions lessen the system; $\exists S_j \in S_{System}, S_j \supset S$. Thus, the new solution can be viewed as a layer on top of the system (like layered-architecture pattern).

The third parameter (ENV) shows how the $\bigoplus$ operator can be achieved and contains environmental factors like the programming language and run-time environment used. We consider these factors because they play an important role in the decision for the technique to be used to evolve the software. For example, if $\bigoplus$ can easily be achieved using run-time techniques then these techniques should be used in evolving the system. This parameter takes values from {RA, CA, In}, where:

- Run-time adaptation (RA): The system provides mechanisms to support the $\bigoplus$ operator, which can be applied at run-time. For example, the system may be programmed with a language that also provides a run-time environment (e.g. a virtual machine). Then the run-time tools provided by the environment can be used to evolve the system.
- Compile-time adaptation (CA): The programming language used has mechanisms that support the $\bigoplus$ operator, such as inheritance and polymorphic calls.
- Installation (In): The addition of new solution to the system is achieved by means of a scripting program, which is used for configuring the system.

We define a context of an evolution problem to be a triple (CHAR,REL,ENV), where CHAR ranges over {E,C, EX}, REL ranges over {NO,O,S,I} and ENV ranges over {RA,CA,In}. Thus, there are 36 contexts for evolution problems. For example, the triple (E, S, CA) denotes the evolution problem in which we want to extend our system using compile-time adaptation techniques to handle evolution requests that specialize the system. Obviously, not all combinations result in a feasible context for an evolution problem. When for a new or changed requirement no solution is found the S does not exist and because of this, we cannot find the intersection of S with the S_{System}. Thus, the contexts (Ex, x, y), where x means any value for REL and y means any value for ENV, are infeasible and there are 24 feasible contexts for evolution problems.

In the PDA input and storage example given in section III, we solved the requirement of supporting encrypted file operations by introducing the class *EncryptedFileWriter*. To combine this class with the system, we need to find the the contexts of this evolution problem. The CHAR parameter should be C (composition) because S is not empty. The intersection of S and the solutions of the system is an empty set, so REL is NO (non-overlapping). We can achieve this composition using compile-time adaptation since we used an object-oriented language. However, if the system that employs our storage provides run-time adaptation or installation techniques, then we can also achieve this composition using run-time adaptation. As a result, we have three contexts for this evolution problem; {C,NO,CA}, {C,NO,In} and {C,NO,RA}.

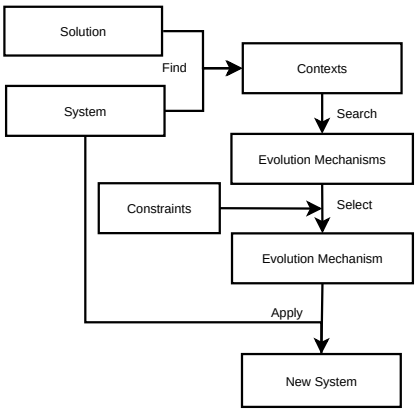

Fig. 2. The function diagram of applying the taxonomy. The arrows are the functions and the boxes are the inputs to these functions

A. Using the Taxonomy

The steps of applying the taxonomy are: solving the new requirement using some design process, identifying the contexts of the evolution problem for the new solution, S, and the system, and then finding the applicable mechanisms these contexts from the list provided in section V.

In Figure 2, we present the functional model for applying the taxonomy. In this figure, the boxes are sets and the connectors are the functions. The starting points of the connectors are the inputs of the function and the end point (the points marked with arrow heads) is the output. The function *Find* refers to the activity of finding the triple context for the evolution problem faced. The sets *System* (referring to the set S_{System}) and *Solution* (S) is required to find the context of the evolution problem. With the *Solution* the characteristic parameter is identified. The *System* is required to identify the environment constraints. Both sets are required to identify the relation parameter. When scenarios are being used to extend the system, the impact of the scenarios is used to identify the relationship between the *System* and *Solutions*. For example, the scenarios may show that a method of a class requires changing, which means the new solutions are overlapping with the system. As discussed in section III, in order to identify this parameter, we need to find the intersection of the solution and the S_{System}. This greatly depends on the components contained in the solution sets. The output of the *Find* function

is the set *Contexts* whose elements are one or more contexts listed in section IV. The function *Search* involves extracting the applicable *Evolution mechanisms* from the list provided in section V. Obviously, not all of the applicable mechanisms can be applied to evolve the system. The system may have some constraints (e.g. memory usage) which may prevent the designer from using some of these mechanisms. The function *Select* refers to the activity of selecting the most applicable evolution mechanism. To select this mechanism, the set *Constraints* is required, which includes the constraints or limitations of the system. After selecting the most applicable mechanism, it is applied to the *System* which evolves the system to *New System*. This procedure is repeated until all new requirements are added to the system.

V. Software Evolution Techniques

In this section, we list the mechanisms, extracted from the software engineering domain, that can be used to address the evolution problems within the 24 contexts given in the previous section.

- {C,NO,CA}: In this context S (the new solution) and old solutions do not intersect and we want to combine them by using compile time mechanisms. For this, we can replace the object that receives the message using polymorphic calls. Or we may want to add new behavior to the existing classes using the observer, composite or the decorator design pattern.
- {C,NO,RA}: In some situations, it may be cheaper to use an already implemented solution rather than re-implementing the solution. Furthermore, the source code of the new solution may not be available, so a run-time adaptation mechanism is required. For such cases, a glue code, which is a dedicated program that replaces or binds the interfaces or modules, can be used.
- {E,NO,CA}: Scenario-based analysis may show that solutions that do not overlap with the current system are going to be added to the system in the future. The mediator pattern provides a class, the mediator, that is the combination point of the other classes. For evolution, the system can be designed using the mediator pattern so that new non-overlapping solutions can be bound to the system by just modifying the mediator.
- {E,NO,RA}: We may want to be able to add non-overlapping solutions to system at run-time. For this, the system can be designed with hook methods (methods without implementations) and the new solutions can implement this hook methods. The best example of this can be found in the plug-in support of web browsers. The main functionality of these browsers is to parse and display web pages. Though, with plug-ins new solutions (e.g. movie player) that extend this functionality can be added to the browsers.
- {C,S,CA}: In this context, the new solution specialize a solution in the system and we would like to compose it with the system by compile time adaptation. The inheritance mechanism supported by object-oriented languages can be used in this context because it supplies transitive reuse. The new solution can inherit existing solutions and add the extra functionality by overriding their methods. We can also compose the new solutions by building a delegation mechanism using the command pattern. The concrete command receivers may aggregate existing solutions or new solutions and the switcher can aggregate these concrete command receivers. The decorator pattern can also be used to extend the functionality of the existing objects.
- {C,S,RA}: If the run-time environment supports editing meta-level dispatcher (e.g. Smalltalk [12] run-time environment) then the solutions that specialize the system can added to the system by modifying this dispatcher. For example, one may want to add new functionality on top of the old functionality to the methods of a class. The "extended" methods that has this new functionality can be implemented in another class and the dispatcher can be modified so that calls are forwarded to this class.
- {E,S,CA}: Analysis may show that in the near future, the functionality of the existing solutions is going to be extended. To ease these future operations, the software engineers can build a call forwarding mechanism by using the bridge or strategy pattern. To use the bridge pattern, for example, the functionality that is going to be extended can be placed in classes that extend the implementor and the users of this functionality should be placed in classes that extend the abstraction (refined abstractions). Then, the client can pass an instance of the functionality (a concrete implementor) to these classes. New functionality can be added by adding a class that extends the implementor and implements the new functionality. Then the client code is also changed so that it passes the refined abstractions to this new class.
- {E,S,RA}: In this context, we want to extend the initial system because we want to be able to specialize the system at run-time. The run-time environments that support this operation may not be suitable for our system; thus we must design our own run-time environment. To only support specialization at run-time, we only need to design a modifiable dispatcher. However, an interpreter that interprets the system can also be designed.
- {C,O,CA}: In some cases, the new solution may require some parts of the system to be changed. For example, the implementation or the interface of a method in the system may be changed. Such changes can be achieved either by reprogramming those parts or using inheritance to override the parts that need to be changed. The adapter pattern can be used to overcome impacts of interface changes. On an interface change, some parts of the system may still require to access the changed components through the old interface. Thus these parts can access the new interface through the wrappers provided by the adaptor. We can use the decorator pattern to replace the behavior of the objects in the system.
- {C,O,RA}: Here, the new solution overlaps with the old

solutions and we want to replace them. The glue code used for gluing non-overlapping solutions can also be used to replace overlapping solutions.

- {E,O,CA}: Using the scenarios, the designers may foresee that in the near future the implementations of the existing solutions are going change. For such cases, the system can be designed to make use of the bridge pattern. Thus the implementations to evolve the system can be changed by just sub-classing the implementor interface.
- {E,O,RA}: As discussed earlier, it may be impossible to stop and make the changes to some systems. To support evolution, these systems are required to be designed in an environment that allows components of the system to be changed at run-time. For example, the Smalltalk [12] object-oriented environment supplies both programming and run-time environments. With this run-time environment it is possible to make modifications to classes. Thus, to support evolution for these systems, the designer may choose to use the Smalltalk environment to develop the initial system.
- {C,I,CA}: In this context, the new solution reduces a solution in the system and we want to combine it with the system using compile time adaptation. The command pattern used for specializing the system can also be used to interpret the system; for example, the concrete command implementors would call some of the functions of the system.
- {C,I,RA}: If the run-time environment of the system supports reflection then it can used to reduce the behavior of a solution in the system. In this way, the new solution can select the methods they are going to use.
- {E,I,CA}: The system can be designed so that the number of its features can be reduced. The layered architectural pattern, for example, can be used while designing the system so that new solution can be placed on top of the existing solutions. Application generators can also be used for this context. Application Generators are compilers that are specifically designed for a purpose (domain specific) [13]. The input to the Application Generator is the program specification and the output is the generated application. Thus, by reducing/changing these specifications we can reduce the systems functionality.
- {E,I,RA}: In this context, we want to design the system in a runtime environment that will allow us to reduce the functionality of the solutions of the system. Thus, we need a runtime environment that supports reflection or we can design the system with reflective architectural pattern to implement such a runtime environment.
- {E,NO,In}, {E,O,In}, {E,I,In}, {E,S,In}: It may be impossible for some systems to stop and to install the system with new solutions. Thus, the software engineer needs to design an environment for the initial system that supports run-time adaptation. The software engineer can design an interpreter or an installation program that configures the modules and call patterns according to a configuration script. So, new solutions can be bound or replaced with the existing solutions.
- {C,NO,In}, {C,O,In}, {C,I,In}, {C,S,In}: In these contexts, we want to add the new solutions to the system or replace existing solutions with the tools provided by the installation system. To achieve this, we need to have an installation system or an interpreter with a configuration script. Thus, we can remove/add solutions to system by changing this script.

For the PDA input and storage system example given in Section III, the context of the evolution problem of adding the encrypted file operations to the system is {C,NO,CA}, {C,NO,In} and {C,NO,RA} as shown in Section IV. Assume that we want to use compile time mechanisms to evolve our system. The destructive solution to this evolution problem is to add the class *EncryptedFileWriter* to the system, which forces the client to use a different interface for encrypted write operations. Thus, components that are going to use encrypted write operations are changed when the *Combine* operations is restarted. Looking at the list provided above, we can see that the mechanisms for this context are polymorphic calls, decorator pattern or observer pattern. To show how the design patterns can be used for this context, we apply the decorator pattern to this evolution problem. In Figure 3, the *StandardFileWriter* class is a simple wrapper that forwards the calls to *FileWriter* class. The *EncryptedFileWriter* class is the decorator and *SharedKeyEncryptedFileWriter* class is the concrete class that implements the shared key encryption operation.

Pros: With this mechanism, we are able to add the support for shared-key encrypted file writing without decomposing the system. Thus, the original design of the system is not affected by the additions. The mechanism, also allowed us to use the same *write* interface for encrypted file writing. Additionally, support for other encryption mechanisms (e.g. public-key encryption) can easily be added to the system by just extending the *EncryptedFileWrite* class.

Cons: To add the encryption operation we had to introduce 3 classes, which has performance and memory draw-backs.

VI. Related Work

There is a substantial body of work on understanding software evolution and providing tools that can ease the software evolving procedure. In this section we briefly summarize some of the work that provides a taxonomy or tools for software evolution.

With their analysis on evolving software, Lehman et al. [14] constructed laws of software evolution. Subsequently, they extend the laws with data collected from various evolving software and listed tools which are direct implications of these laws [15]. For example, as an implication of the conservation of familiarity law, Lehman suggests collecting and modeling growth data so that this model can later be used in estimating the growth trend per release. We also provide tools to cope with evolution; however, the tools we provide can be used at design time and address the problem of integrating new requirements to the system.

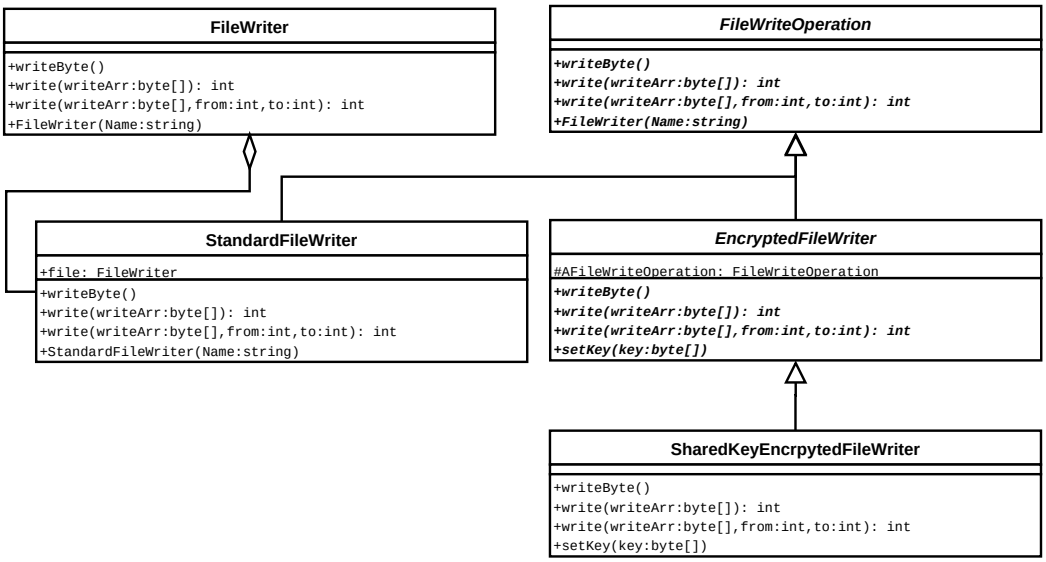

Fig. 3. The system after the addition shared key encrypted file operation using the decorator pattern.

Chapin et al. [16] provide a classification of types of software evolution activities such as changing the source code or the documentation. The focus of this taxonomy is different from our taxonomy, since we identify the contexts of the evolution problems and find well established methods that allow constructive evolution of the software.

Perry [1] states that classifying software evolution activities is limiting because the sources of evolution that affect the way systems evolve is not considered. Following this argument, Perry lists the domain, experience and process as the sources of software evolution. We base our taxonomy on the fact that a change in one of these sources has occurred or is expected to occur. Thus, given an evolution problem, our taxonomy can be used to find the mechanisms that are applicable to it. Buckley et al. [17] provide a taxonomy for evolution that also focuses on the factors that affect the mechanisms that can be used to evolve the system. The main difference between their taxonomy and ours is that we view evolution as an integration of new requirements to the existing system and we use this view to extract the factors.

Refactoring refers to the activity of changing the structure of a program without affecting its external behavior [18]. The aim of such changes is to increase the quality of software. When applied correctly, for example, they can increase the extensibility of the software [19]. Some of the mechanisms we provide in this paper can also be considered as refactorings since they increase the extensibility or modifiability qualities of software without changing the behavior of the system. Besides these, we also provide mechanisms, which can be used easily to change the behavior of the software.

VII. Conclusions and Future Work

In this paper, we considered the software evolution problem as an integration process in which new solutions are added to the system and we listed the mechanisms that allow evolution with out breaking down the software. To list these mechanisms, we first identified the types of changes that may occur in requirements. Then, we build a model for evolution, where the solutions for the changed requirements were composed to the existing system to give the new system, the system we want to achieve. For integration, we concluded that three parameters have an effect on the set of applicable evolution mechanisms, which are the characteristic of the new solutions (CHAR), the impact of the new solutions on the existing system (REL), and the environment in which the existing system runs (ENV). We presented the context of an evolution problem as a triple (CHAR,REL,ENV). According to the values these parameters get, we identified 36 contexts. We reduced the contexts by doing feasibility analysis and in the end we identified 24 feasible context for evolution problems. We concluded our discussion by providing some well established mechanisms that are suitable for the identified contexts.

Applying constructive approach to software evolution requires identifying the contents of solutions sets and finding the intersection of these sets. In this paper, we showed general idea of constructive approach; however, in order to apply the approach we need the what is changing aspect of evolution, which we call the perspective. The perspective, defines the models that are elements of the solution sets. With the PDA input and storage example (presented in Section III), we introduced the *Design* perspective, thus the models used are object-oriented software components. The perspective used also has an effect on the evolution mechanisms. For the design perspective, the changing aspect are components like classes, thus we selected the mechanisms that allow flexibility at this perspective. If we were to use *implementation* perspective (which deals with implementation details of methods) then the mechanisms used should support constructive evolution at this level. In subsequent papers, we will detail different perspec-

tives, their models and the mechanisms for each perspective.

It is possible for an evolution problem to have more than one context and, in turn, more than one solution. Thus, besides identifying the context of the problems, the trade-off between different evolution techniques should also be considered in selecting a technique. For example, when performance is considered an important quality of a system then certain techniques may not be used to evolve the system. Noppen, Van den Broek and Aksit [11] represent requirements as fuzzy sets to overcome vague and conflicting information. In this study, the requirements are associated with degrees like user satisfaction. Using a design process the solutions to these requirements are found and each solution is associated with a cost. Then the trade-off between user satisfaction and cost is analyzed. We are going to apply the fuzzy set representation to alternatives between evolution mechanisms to show the trade-off between these alternatives.

In this paper, we have build the taxonomy for only integration evolution problem. Though, due to evolution the requirements can be modified or removed. For integration problems, we have identified the contexts of evolution problems by looking at the relationship between the new solution and the solutions of the software system. A similar approach can also be used for modification and removal. First we need to find the solutions to be modified or removed in the system. Then, by looking at the interaction of these solutions with the remaining solutions of the system we can categorize their relationship. Here, the interactions are the intersection of the solution sets; thus, we can say that the categories of the changes for modification and removal is similar to the categories for integration. Using our taxonomy as basis, we will extend the list of mechanisms to cover modification and removal.

REFERENCES

[1] D. E. Perry, "Dimensions of software evolution," in *Proceedings of the International Conference on Software Maintenance* (Victoria, B.C., Canada; September 19-23, 1994), H. A. Müller and M. Georges, Eds., 1994, pp. 296–303.

[2] B. Tekinerdogan, "Synthesis-based software architecture design," Ph.D. dissertation, University of Twente, Mar 2000.

[3] L. Dobrica and E. Niemel, "A survey on software architecture analysis," *IEEE Transactions on Software Engineering*, vol. 28, pp. 638–653, July 2002.

[4] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Addison-Wesley Professional, 1999.

[5] R. Kazman, L. Bass, G. Abowd, and M. Webb, "Saam: A method for analyzing the properties of software architectures," pp. 81–90, 1994.

[6] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, "Architecture-level modifiability analysis (alma)," *Journal of Systems and Software*, vol. 69, no. 1-2, pp. 129–147, 2004.

[7] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," *iceccs*, vol. 00, p. 0068, 1998.

[8] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta, "A controlled experiment in maintenance comparing design patterns to simpler solutions," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1134–1144, 2001.

[9] T. Mens and A. H. Eden, "On the evolution complexity of design patterns," *Electronic Notes in Theoretical Computer Science*, vol. 127, no. 3, pp. 147–163, April 2005.

[10] K. H. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE - Future of SE Track*, 2000, pp. 73–87.

[11] J. Noppen, P. van den Broek, and M. Aksit, "Dealing with fuzzy information in software design methods," in *2004 Annual Meeting of the North American Fuzzy Information Processing Society*, S. Dick, L. Kurgan, P. Musilek, W. Pedrycz, and M. Reformat, Eds., vol. 1. Institute of Electrical and Electronics Engineers, Inc., 2004, pp. 22–27.

[12] A. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1993.

[13] E. Horowitz, A. Kemper, and B. Narasimhan, "Application generators: Ideas for programming language extensions," in *ACM 84: Proceedings of the 1984 annual conference of the ACM on The fifth generation challenge*. New York, NY, USA: ACM Press, 1984, pp. 94–101.

[14] L. Belady and M. Lehman, "A model of large program development," *IBM Sys. J.*, vol. 15, no. 1, pp. 225–252, 1976.

[15] M. M. Lehman and J. F. Ramil, "Rules and tools for software evolution planning and management," *Annals of Software Engineering*, vol. 11, no. 1, pp. 15–44, 2001.

[16] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *Journal of Software Maintenance*, vol. 13, no. 1, pp. 3–30, 2001.

[17] J. Buckley, T. Mens, M. Zenger, A. Rashid, and G. Kniesel, "Towards a taxonomy of software change: Research articles," *J. Softw. Maint. Evol.*, vol. 17, no. 5, pp. 309–332, 2005.

[18] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[19] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.