

A Case Study of Unanticipated Incremental Change

Václav Rajlich, Prashant Gosavi

Department of Computer Science, Wayne State University, Detroit, MI 48202

rajlich@cs.wayne.edu

Abstract

Incremental changes add new functionality and properties to software. They are the core of software evolution, maintenance, iterative development, agile development, and similar software processes. This paper presents a technique for unanticipated incremental software change and a case study. The technique uses programming concepts as the foundation, and contains steps of concept location, actualization, incorporation, and change propagation. The case study presents an unanticipated change in the open source software “Drawlets”.

1. Introduction

Incremental software change is an essential part of software evolution, maintenance, iterative development, iterative enhancements, agile development, Extreme Programming, and similar software processes [9]. Each incremental change adds new property to software not available in earlier versions.

Current research has focused on refactoring, which changes the structure of the software but leaves the functionality unchanged [7]. There is also sizable literature on software design that anticipates future changes and preparing software for them [11]. However, very often and for variety of reasons, a change is not anticipated and hence the software is not prepared for it. Unanticipated incremental change is a substantial part of software engineering practice.

In the literature, there are few articles that deal with unanticipated incremental change. Unanticipated incremental change is still a self-taught art that each programmer has to acquire through trial and error, a very costly method. This paper attempts to fill that void and presents a technique for unanticipated incremental change and a case study. The technique emphasizes the role of software comprehension and the role of programming concepts.

Section 2 presents the technique and explains the role of the concepts. Section 3 describes the case study. Section 4 is a continuation of the case study and describes additional techniques that shorten the change propagation. Section 5 summarizes the case study experience. Section

6 relates this work to other research, and section 7 contains conclusions and future work. Appendix contains further details of the case study.

2. Unanticipated incremental change

In this section, we present a technique for an unanticipated incremental change. At the core is a simple observation: Requests for incremental software change are formulated in terms of program concepts and therefore the change technique has to focus on program concepts. An example of a change request for a class registration program is: “Before registering a student for a class, add a check of prerequisites”. The process of the change starts with concept location, where the code that implements the relevant concept (“register”) is located in the software. Once the location of the concept is found in the code, it will become the center of the software change.

Since most of the change requests originate from the end user, the user’s interaction with the program is the source of most of the concepts that govern the change. Among them, domain concepts are often used as the starting point (“register”). Beside them, there are other concepts related to various aspects of user’s interaction with the system (“network error while validating registrant’s ID”). There also may be change requests originating from other programmers that deal with design concepts (“composite pattern for the transcript”).

The programmer starts the change by locating relevant concepts in the code. The concepts are sometimes present in a primitive form and have to be expanded. For example, the concept “register” is already in the program but has to be broadened by adding the check of prerequisites. Other times, concepts are implicit in the assumptions on which a code segment is based. For example, another way how to understand the same change request is to conclude that in the old student registration program, the fact that the check of prerequisites is not made is equivalent to the assumption that the prerequisites are always satisfied. The code segment based on this assumption is the location of the “prerequisites” concept. The latter type of concept is called *latent* concept.

In order to make the change, the programmer locates the concepts (whether primitive or latent) and then implements them explicitly and fully. The technique for

unanticipated incremental changes consists of the following steps:

- *Request*: Formulate the change request.
- *Extraction*: Extract the applicable concepts from change request.
- *Location*: Locate the concepts in the code.
- *Actualization*: Implement the concepts explicitly and fully in new code.
- *Incorporation*: Replace the latent concepts in the old code by program dependencies between the old and new code ("plug in" the new code).
- *Propagation*: Propagate change through the old code and fix all inconsistencies introduced by the change.

Please note that implicitly in this technique, it is not necessary to understand the whole code of the program, but it is sufficient to understand only the parts involved in the individual steps. Also please note that the steps are not always sequential, that there may be backtracks where programmer may start with different but related concepts, skipping trivial steps, or merging two or more steps into one.

The rest of this section explains the technique in more detail.

2.1 Location

Concept location was studied as a topic of its own and there are several methodologies available, see the overview in [10]. In our case study, we followed the technique of [4] and located the concepts by the search of static code. The search space is the class diagram and the programmer conducts the search by going top-down through control flow dependencies.

When using this technique, the programmer first visits the top class of the program where the functionality of the whole program is summarized. However this top class does not (and cannot) do everything, it delegates parts of its functionality to other classes. Hence if the top class does not implement the sought concept, it must be implemented by one of the classes called by it. Since these classes are specialized, it is usually easy to decide which one does and which one does not contain the sought concept. Moving down through the call graph towards more and more specialized classes, the programmer ultimately finds the classes that participate in the concept.

2.2 Actualization and incorporation

Actualization implements the enhanced concept, usually as several new classes. Incorporation then interconnects these new classes into the old code. That can be done by declaration of the instances of the new classes in the old code, or method calls from the old code to the new or vice versa.

2.3 Change propagation

If class C changes, all classes interacting with C may have to change also. It is obvious that a change in the class interface forces changes in the classes that use the interface of C, but more subtle changes also propagate. For example, while the interface of a public method remains the same, the change in its algorithm may require changes in the context of the call. Therefore the programmer should visit all interacting classes after every change [8].

The issue of likelihood of change propagation is often raised in this context. Different kinds of interactions among the classes have different likelihood of change propagation. For example, a change in a derived class is less likely to trigger a change in the base class, while a change in a base class is more likely to require changes in derived classes. However, a safe view is that all classes that have non-zero probability of change must be visited and either changed, or certified that they do not need change.

There is also the issue of hidden propagation [14]. Sometimes information passes from class A to class B through an intermediary class X. Then a change in A may trigger a change in B, while the intermediary class X remains unchanged. An example of this situation is in [14] and the case study in the following section also has an example of hidden propagation. The existence of hidden propagation has the following consequence: If in the process of change propagation we encounter a class that does not change, we still have to investigate whether there is a potential for hidden propagation. If there is, we have to visit the neighbors of the class even if the class itself did not change.

2.4 Granularity

Another issue is the granularity of the steps. In this paper we emphasize the granularity of classes, where the program concepts are most clearly expressed and many of them are implemented as classes. Classes are also natural units of code for code editors and software tools.

Compared to that, smaller granularities provide information that is more accurate but may be too detailed and voluminous. As such, it may be appropriate only for small changes. Some changes may operate on a combination of granularities, for example, the location done on the granularity of classes, actualization and incorporation on the granularity of class members, and propagation on the granularity of statements.

2.5 Special cases

The technique described in this section is specifically suitable for large and de-localized changes. Changes that are either small or localized still can use it, but some steps may become trivial. For example, in a small or familiar program, concept location is immediate and does not require any explicit step.

Localized changes are contained within a single class and avoid change propagation. For small changes it may be reasonable to merge actualization and incorporation and change the old code directly without separating these two steps, as done in the case study of [4]. Changes that retract functionality replace actualization and incorporation by deletion of the obsolete code, while location and propagation still may be necessary.

2.6 Shorter propagation

One of the difficult issues of an unanticipated change is change propagation. If the program architecture does not support the change, change propagation can affect many classes. A change in any class increases the risk that something in the code will break, and hence an attempt to shorten the change propagation is a very reasonable effort. Two techniques for shortening the change propagation are part of our case study.

The first technique is *refactoring* [6], [7]. Opportunistic refactoring prepares the software for the change by grouping code affected by change into fewer classes. After the refactoring is done, those classes will be the only classes affected by the change, and the change propagation will be shorter.

The second method is *splitting of roles*. Often in the code, the same class or same method is used in two slightly different roles. Since both roles are very similar, the same code can do both jobs.

The propagating change however may highlight the differences in these two roles, when only one of them needs to be updated while the other one can stay the way it was before. Splitting the two roles and updating only one of them shortens the propagating change. This is illustrated in our case study in Section 4.

2.7 Unit tests and software documents

In this paper we do not explicitly discuss the unit tests [1] and software documents like manuals and documentation, but the change presented here is compatible with them. All steps, including change propagation, can be applied to the production code, the unit tests, and software documents. Change propagates through the dependencies among the documents.

The next section presents a case study of an unanticipated de-localized change.

3. Case study

We conducted a case study of the unanticipated incremental change on framework “Drawlets” [5], [15]. One of the authors conducted the change, while the other author was an observer. It was a revelatory case study and the purpose was to assess the suitability of the technique for unanticipated incremental change.

Drawlets is a framework that adds graphics to a host application. Kent Beck and Ward Cunningham developed the original framework named “HotDraw” in Smalltalk. RoleModel Software then ported it into Java and renamed it Drawlets.

The main feature of Drawlets is the drawing canvas that holds figures and allows users to interact with them. Tools modify the attributes of a figure such as size, and location. A special selection tool selects multiple figures and modifies all of them.

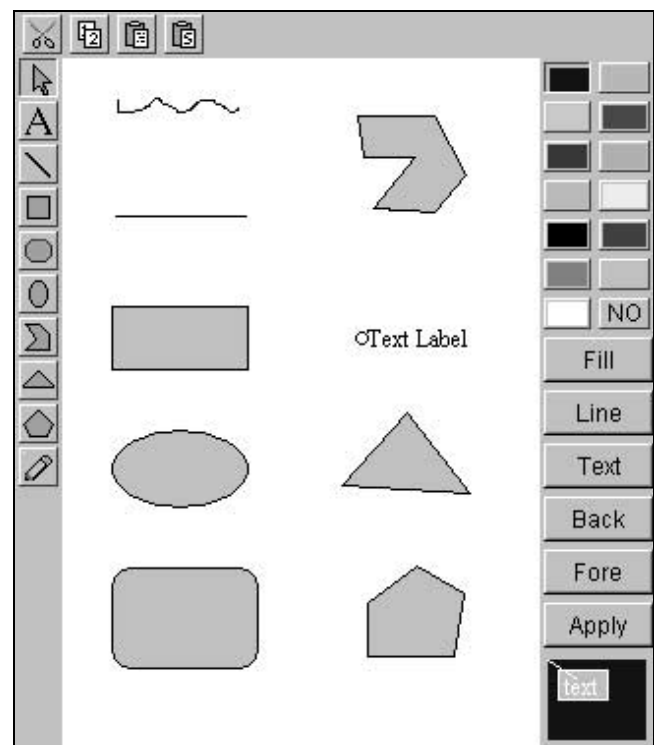


Figure 1: Sample screen shot of SimpleApplet

Drawlets supports lines, free-hand lines, rectangles, rounded rectangles, triangles, pentagons, polygons, ellipses, and text boxes. Each figure has a separate drawing tool. For example, clicking on the line tool button and then clicking on two different points on the canvas will create a line between the two points.

Drawlets has more than 100 classes, 35 interfaces and 40,000 lines of code and documentation.

The host application provides an instance of the drawing canvas, toolbars, and tool buttons. In this case study, we used a host application called SimpleApplet. Class SimpleApplet implements this host application and is a part of the Drawlet library [15].

SimpleApplet runs in any browser that supports Java. It displays the drawing canvas in a browser window and arranges the toolbars and tool buttons around it. These tool buttons activate tools that create or modify figures. Figure 1 shows the SimpleApplet layout and a selection of different shapes it can draw.

Two toolbars are located on the top and left sides of the canvas and each button in the toolbars activates one tool or command. A tool palette is located on the right side of the canvas and modifies the attributes of figures. Figure 2 shows the UML class diagram [3] of select high-level classes in SimpleApplet application.

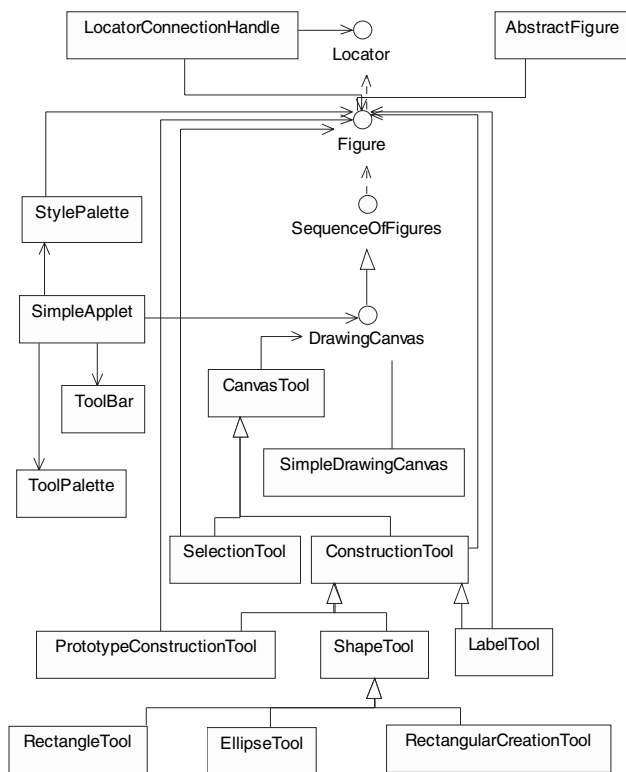


Figure 2. Top classes of SimpleApplet application

3.1 Change request

The change request is to implement an "owner" for each figure. An owner is a user who put that figure onto the canvas, and only the owner is allowed to move and modify it. At the beginning, each session declares a session owner, and this session owner will own all new

figures created in that session. No other user will be allowed to manipulate them. At the beginning of a session, user inputs ID and password. Any function that attempts to move a figure must validate that the figure owner and the current session owner are same.

This change will make SimpleApplet more versatile and useful. Users will be able to work on a single document collaboratively and the other users will not be able to modify their figures.

The first step is the extraction of relevant concepts from change request.

3.2 Extraction

The concepts relevant to the change are the figure owner and session owner. Both of them are implicit, the old code assumes that there is just one owner who owns both sessions and figures.

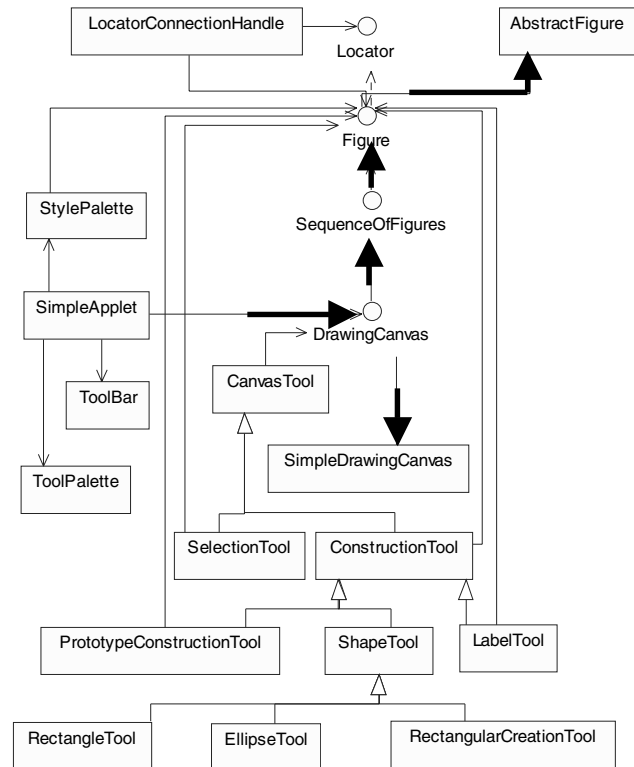


Figure 3. Summary of concept location

3.3 Location

In order to locate both concepts, we followed search technique explained in the previous section and in [4]. The search is illustrated in Figure 3 and it started in the class SimpleApplet.

Class SimpleApplet is responsible for the whole application. However it does not contain either of these concepts and therefore they must be located in the classes used by it. StylePalette, as the name suggests, is a palette of styles that can be applied to figures selected by the selection tool. We concluded that it couldn't contain the concepts. Similarly, the concepts are not located in Toolbar or ToolPalette and therefore they must be located in DrawingCanvas or the classes used by it.

Class SimpleDrawingCanvas implements interface DrawingCanvas. There is only one instance of SimpleDrawingCanvas in the whole application. Hence it is the right place to store the current session owner and we concluded that the implicit current session owner is located there.

Then we searched for the location of the concept of figure owner. Since DrawingCanvas does not contain it, we visited class SequenceOfFigures. This class is just a collection of Figure objects and therefore does not contain the concept of a figure owner. Next we visited interface Figure. Class AbstractFigure implements the interface Figure.

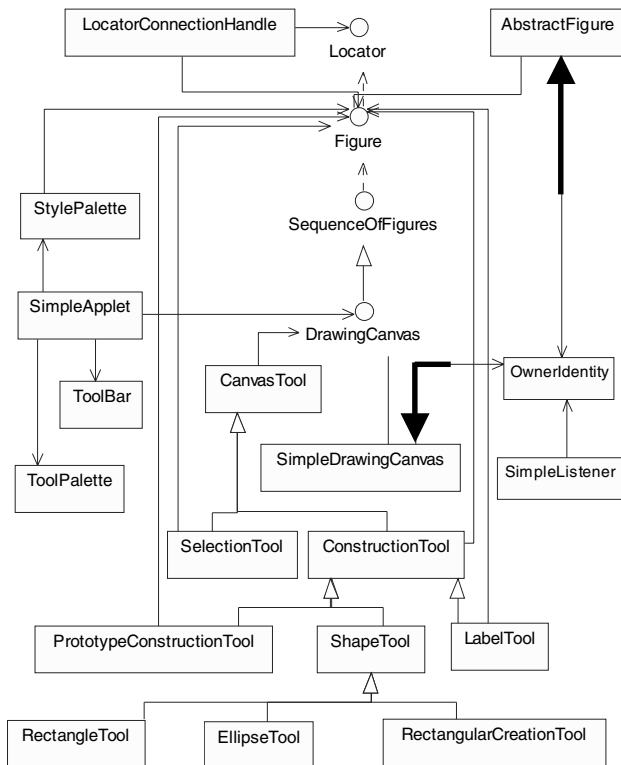


Figure 4: Situation after actualization

Class AbstractFigure holds most of the attributes of a Figure and implements a majority of the functions defined in interface Figure. It also is the root class of the class hierarchy of the shapes. We concluded that the implicit

concept of figure owner is located in class AbstractFigure. The summary of the location process is in Figure 3.

3.4 Actualization

We implemented the concept “owner” in a new class OwnerIdentity. It holds basic owner data, such as owner ID and a password. A dialog box allows users to change current session owner. It also has a listener to listen when the user clicks on a button in this box. We implemented the listener in class SimpleListener.

The architecture of SimpleApplet after actualization is shown in Figure 4. This figure shows two newly added classes OwnerIdentity and SimpleListener, and the marks point to the classes where the concepts were located and therefore have to be changed. The marks indicate the inconsistencies in the program and the direction in which they propagate.

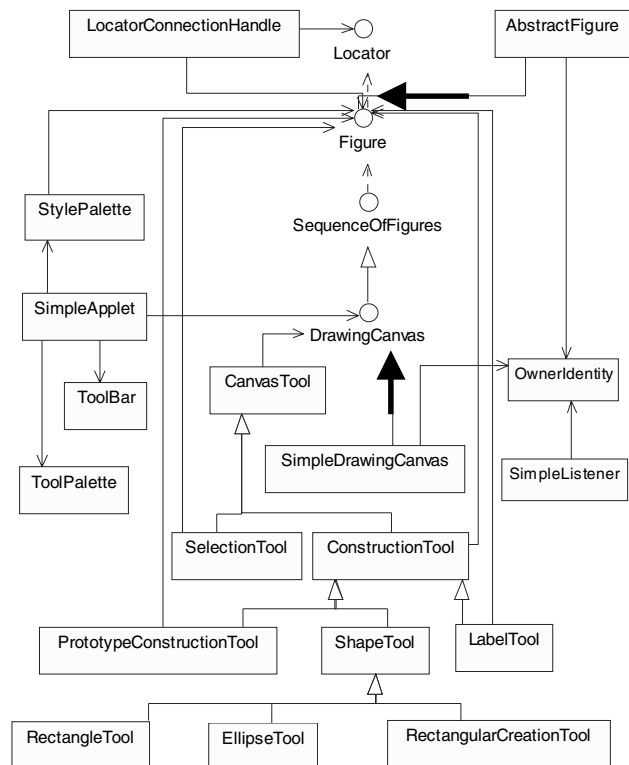


Figure 5: Situation after incorporation

3.5 Incorporation

We incorporated the newly created classes OwnerIdentity and SimpleListener into the old code in two places, both of them found during the concept location. We created an instance of OwnerIdentity in both AbstractFigure and SimpleDrawingCanvas.

In class `AbstractFigure`, we added three new public methods `setFigureOwner()`, `int getFigureOwnerID()` and `String getFigureOwnerName()`. We also modified public methods `move(...)` and `translate(...)` into `securemove(.., int sessionOwnerID)` and `securetranslate(..., int sessionOwnerID)`. Several other functions in this class that called `move(...)` and `translate(...)` were also modified.

In class `SimpleDrawingCanvas`, all functions that call `move(...)` and `translate(...)` of the `Figure` interface were modified to handle the extra parameter.

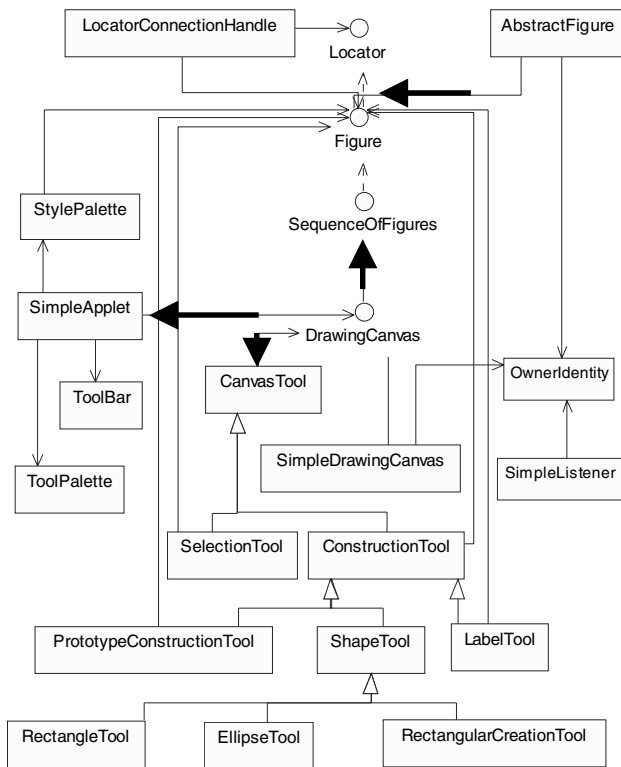


Figure 6: Visiting classes that interact with `DrawingCanvas`

Figure 5 shows the situation after incorporation, where classes `SimpleDrawingCanvas` and `AbstractFigure` were modified. Since both `SimpleDrawingCanvas` and `AbstractFigure` changed, we marked all their interaction with the other classes.

3.6 Change propagation

After the concept actualization and incorporation, several inconsistencies were left in the code. We fixed the inconsistencies by a change propagation process. The first

part of the propagation (Figures 5 through 8) is shown here, the rest is shown in the Appendix.

First we visited classes and interfaces interacting with `SimpleDrawingCanvas`, see Figure 5. The only interface interacting with `SimpleDrawingCanvas` is interface `DrawingCanvas`. The interface itself does not need any change, but it propagates the changes to the classes that interact with it (hidden propagation). We visited them in the next step.

In the next step we visited classes interacting with `DrawingCanvas`, see Figure 6. Of them, class `CanvasTool` and interface `SequenceOfFigures` did not need any modifications and did not propagate change any further. However, class `SimpleApplet` needed a change to call the method `getNewID()`. Therefore we added a new button to the toolbar and new command "Get New ID".

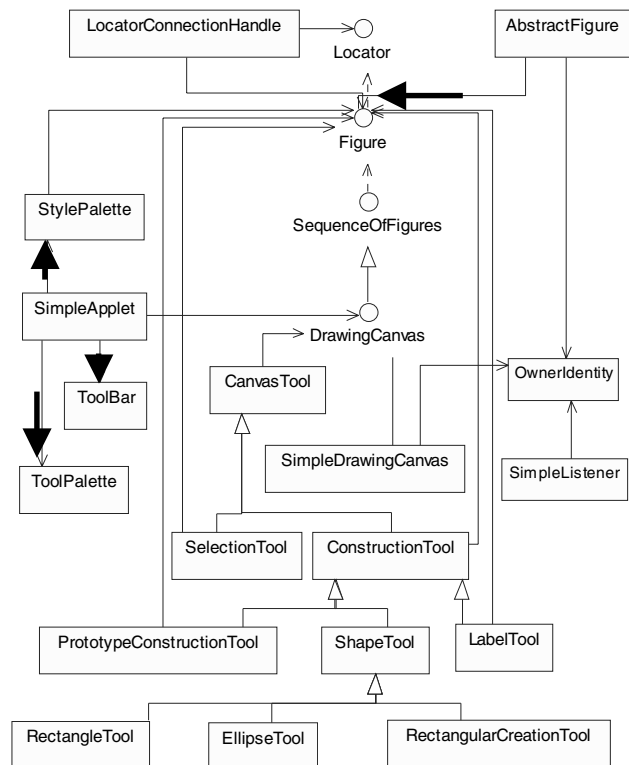


Figure 7: Visiting classes that interact with `SimpleApplet`

In the following step we visited classes interacting with `SimpleApplet`, see Figure 7. Class `SimpleApplet` interacts with `ToolBar`, `ToolPalette` and `StylePalette` classes and hence we visited all three of them, but none of them needed to be modified. They also did not propagate the change any further. The result is shown in Figure 8. Please note that Figure 8 still contains one inconsistency mark, and hence the change propagation is still incomplete. The rest of the change propagation is shown in the Appendix.

4. Shorter change propagation

One of the difficulties of unanticipated incremental change is the length of the change propagation. In the case study, we also explored processes that shorten the change propagation.

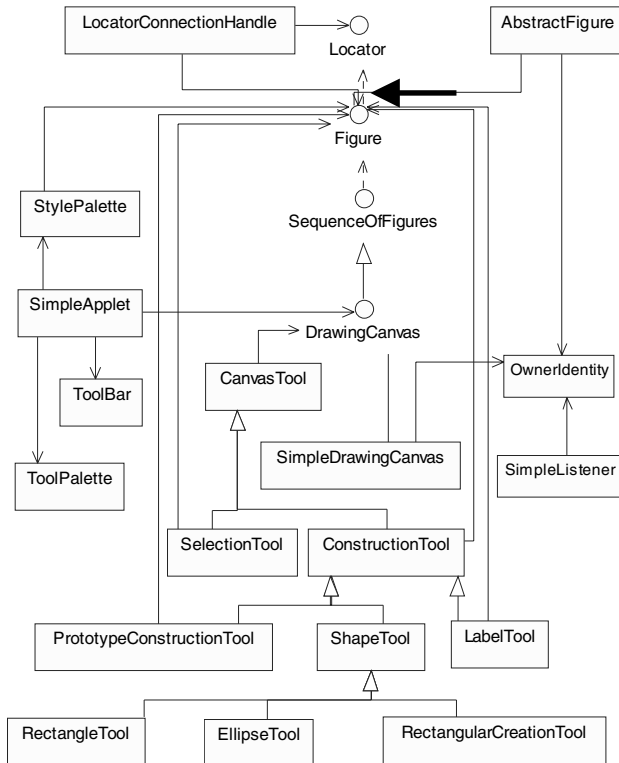


Figure 8: Visiting classes that interact with AbstractFigure

4.1 Refactoring

We noticed each tool that creates new figures has the function `basicNewFigure(...)` that has two main parts:

- a. Create a new figure
- b. If the figure was not created at its intended location, move it.

We refactored `basicNewFigure(...)` by transformations described in [7] and extracted a method called `moveFigure(...)` that moves a newly created figure. We made it a member of the base class `ConstructionTool` and classes that create Rectangles, Ellipses, etc. inherit it.

This refactoring improved the code by removing redundancies and it also substantially shortened the change propagation. In the refactored code, we did not need to visit and change classes `LabelTool`, `PrototypeCreationTool`, `EllipseTool`, `RectangularCreationTool`, and

`RectangleTool`, because changes in `ConstructionTool` now did not propagate further.

4.2 Splitting the roles

We observed that the function `move(...)` is used in Drawlets in two different ways: to move the figure as requested by the user, or as a part of the algorithm for the creation of the new figures. In the second case, the figure is first created in a predetermined location and then moved to the location selected by the user. Figures `Ellipse`, `Rectangle`, etc. are created in this way, while figures `Line`, `Polygon`, etc. are created directly in the final location and hence do not invoke function `move(...)`. The extended change propagation is caused by this dual role of function `move(...)`.

It is obvious that a user-initiated move must check user identity, while the creation of new figures does not. By splitting these two roles into two functions, a new function `secureMove(...)` and at the old function `move(...)`, we substantially shortened the change propagation. The new function `secureMove(...)` first checks the identity of the user and then invokes the function `move(...)`.

Function `translate(...)` is also used to move figures, but it does not play any dual roles. It is called only in response to a user's request, so we simply modified this function without the need to split the roles.

5. Observations

Drawlets software contains an extensive API and it was designed to localize many likely changes. However the architecture of the software did not support the change that we made. Data in Table 1 shows how widely delocalized it is and also shows the impact of the techniques that limit change propagation. Please note that the techniques limit the number of classes to which the change propagates but do not necessarily decrease the number of modified lines.

Table 1: Numerical data for case study.

	propagation	refactoring	splitting
classes added	2	2	2
classes modified	13	8	5
interfaces modified	1	1	1
LOC modified	91	95	87

In our case study, we chose to be cautious and inspect all classes that showed the slightest probability that they may need a change. There is a practical limit how far to carry change propagation, otherwise every class of the

program would be visited. In our case study, we used the following criterion: We visited all neighbors of the classes that have changed. If the class changed, we made the appropriate changes and propagated the change further. If not, we assessed whether it propagates the change to its own neighbors. This criterion worked well in our case study.

The case study also showed that the granularity of classes for this kind of change works well for both concept location and change propagation. It was easy to narrow the search to finer granularity within a class when it became necessary.

6. Relation to other work

An early paper on unanticipated incremental change appears in [13].

Drawlets were proposed as a test-bed for software evolution in [5].

The role of evolution in software lifecycle was summarized in [9]. During software evolution, it is often necessary to make other changes that are not incremental. Refactoring is a change that does not modify the behavior of the program but modifies program structure and makes it more suitable for future evolution. A comprehensive list of refactoring transformations is in [7]. In [6], the authors study refactoring of C++ code.

Concept location has been studied in several papers, starting with [1]. A recent overview of this research appears in [10]. Static concept location is explored in [4] and dynamic concept location appears in [12].

In [8], the author defines evolving interoperation graphs that are used as a theoretical model of change propagation. The notation of [8] is the basis for the figures of Section 3.

Change impact analysis concentrates on finding all classes to be impacted by change. Research of impact analysis was summarized in [2].

7. Conclusions and future work

In this paper, we presented a technique for unanticipated incremental changes. Program concepts play an important part in the technique.

We used this technique in a case study of an unanticipated incremental change in Drawlets, a moderately sized software system written in Java. The original program architecture did not support this change, thus the change was de-localized. We also studied two variants of limiting change propagation, based on refactoring and role splitting, respectively. Role splitting turned out to be a very effective way to limit the number of classes visited during the change propagation.

This paper raises several issues for future work. In order to deal with a backlog of change requests and minimize the amount of rework, we intend to study the optimal sequence of incremental changes that depend on the relationship, coupling, and complexity of the program concepts.

In this paper we worked with the granularity of classes. However the finer granularities offer greater accuracy. Future work will study this impact of granularity on accuracy.

We observed that both concept location and change propagation are an overhead in the change, while actualization and incorporation constitute the core of the change. Small changes lead to multiple repetition of the overhead and we speculate that the result is decreased programming efficiency along with an increased likelihood of errors. On the other hand, changes that are too big and deal with too many issues at once may overload the cognitive capabilities of the programmers. Hence, we speculate that there is an optimal size of the incremental change and we will study this optimal size.

We expect that the research of incremental change will improve the current situation where the incremental change is largely a self-taught art, and will allow the accumulation of knowledge in this important field.

References

- [1] Biggerstaff, T.J., B.G. Mitbender, D.E. Webster, "Program Understanding and the Concept Assignment Problem", *Communications of ACM*, May, 1994, 72-78
- [2] Bohner, S.A., Arnold, R.S. Software Change Impact Analysis. *IEEE Computer Society Press*, Los Alamitos, CA, 1996.
- [3] Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language User Guide. *Addison-Wesley*, Reading, MA, 1998.
- [4] Chen, K., Rajlich, V., RIPPLES: Tool for Change in Legacy Software, in *Proceedings of IEEE International Conference on Software Maintenance*, 2001, IEEE Computer Society Press, 230 – 239
- [5] Demeyer, S., Mens, T., Wermelinger, M., Towards a Software Evolution Benchmark, *Proceedings of International Workshop on Principles of Software Evolution – IWPSE 2001*, to be published by ACM Press.
- [6] Fanta, R., Rajlich, V. Reengineering an Object Oriented Code. In *Proceedings of IEEE International Conference on Software Maintenance*, 1998, IEEE Computer Society Press, 238-246.
- [7] Fowler, M. Refactoring: Improving the Design of Existing Code. *Addison-Wesley*, Reading, MA, 1999.

[8] Rajlich, V. Modeling Software Evolution by Evolving Interoperation Graphs, In *Annals of Software Engineering*, Vol. 9, 2000, 235-248.

[9] Rajlich, V.T., Bennett, K.H. The staged model of the software lifecycle. *IEEE Computer*, July 2000, 66-71.

[10] V. Rajlich, N. Wilde, The Role of Concepts in Program Comprehension, Proc. IEEE International Workshop on Program Comprehension, IEEE Computer Society Press, 2002, 271 – 278.

[11] Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, 1972, Vol. 29, pp.1053-1058.

[12] Wilde, N., M.C. Scully, "Software Reconnaissance: Mapping Features to Code", *J. Software Maintenance*, 1995, 49-62.

[13] Yau, S.S., Collofello, J.S. and MacGregor, T. (1978), "Ripple Effect Analysis of Software Maintenance," In *Proceedings of CompSAC*, IEEE Computer Society Press, Los Alamitos, CA, pp. 60-65.

[14] Yu, Z., Rajlich, V., Hidden Dependencies in Program Comprehension and Change Propagation, In *Proceedings of 9th International Workshop on Program Comprehension*, 2001, IEEE Computer Society, 2001, 293 – 299

[14] <http://www.rolemodelsoft.com/aboutUs/drawlets.htm>

Appendix

Please note that in Figure 8, there is still one inconsistency mark left. This Appendix lists the remaining four steps of change propagation that deal with that inconsistency.

Step 1: Visit to classes interacting with class AbstractFigure.

We visited Figure interface and added public functions `setFigureOwner(String s, int ID)`, `getFigureOwnerID()`, and `getFigureOwnerName()`. Also, the `move(...)` and `translate(...)` functions must be modified to include an extra parameter. We renamed these functions `securemove(..., int currentSessionOwnerID)` and `securetranslate(..., int currentSessionOwnerID)`. The situation after this change is shown in Figure 9.

Step 2: Visit to classes interacting with interface Figure, see Figure 9.

- Interface Locator did not need any modifications and did not propagate change any further.
- Class StylePalette calls `setFigureOwner(...)` and `move(...)` functions of the Figure interface. The `move(...)` function has been renamed as `securemove(..., int currentSessionOwnerID)` and an extra parameter was added to it. Hence, modification was required in class

StylePalette. Neighbors of StylePalette did not need any modifications and the change did not propagate in this direction.

- Class LocatorConnectionHandle calls `move(...)` function of the Figure interface, hence modification is required. Neighbors of LocatorConnectionHandle did not need any modifications and the change did not propagate further.
- Class SelectionTool calls `translate(...)` function of the Figure interface, hence modification is required. Neighbors of SelectionTool did not need any modifications and the change did not propagate further.
- Class PrototypeConstructionTool calls `move(...)` function of the Figure interface, hence modification is required in class PrototypeConstructionTool. Neighbors of PrototypeConstructionTool did not need any modifications and the change did not propagate further.
- Class ConstructionTool calls `setFigureOwner` and `move(...)` functions of the Figure interface. Hence modification is required in class ConstructionTool. Class ConstructionTool calls `setFigureOwner` and `move(...)` functions of the Figure interface. Hence modification is required in class ConstructionTool.

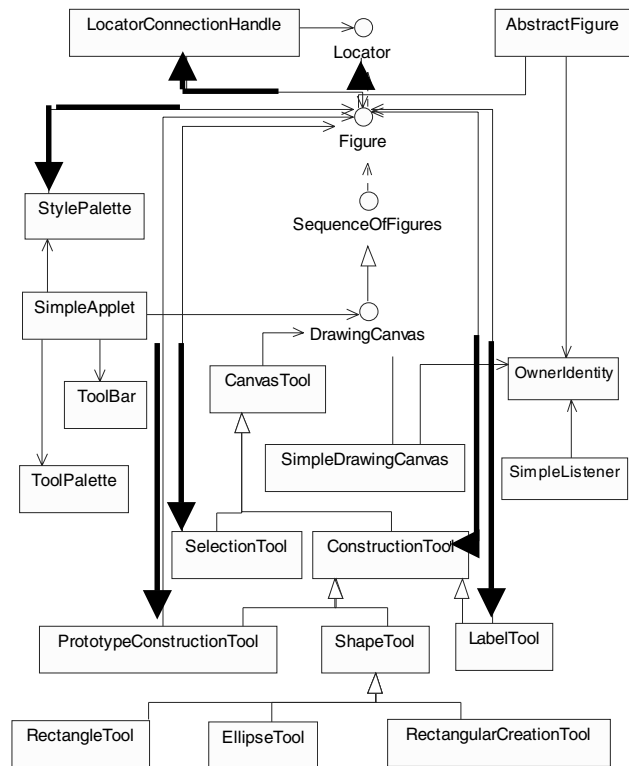


Figure 9: Visiting classes that interact with Figure interface

- Class ConstructionTool calls `setFigureOwner` and `move(...)` functions of the Figure interface. Hence modification is required.

Step 3: Visit to classes that interact with `ConstructionTool`, see Figure 10.

- Class `PrototypeConstructionTool` was already visited before and does not need any change.
- Class `LabelTool` calls `getCurrentSessionOwner()` of class `ConstructionTool` and `move(...)` function of the `Figure` interface, hence modification is required. Neighbors of `LabelTool` did not need any modifications and the change did not propagate further.
- `ConstructionTool` interacts with `ShapeTool` and hence we visited it. Even though no modification was required in this class, it is an abstract class and a number of classes derive from it. So it is possible that these classes needed to be modified (hidden propagation). Hence, we visited classes that interact with `ShapeTool`.

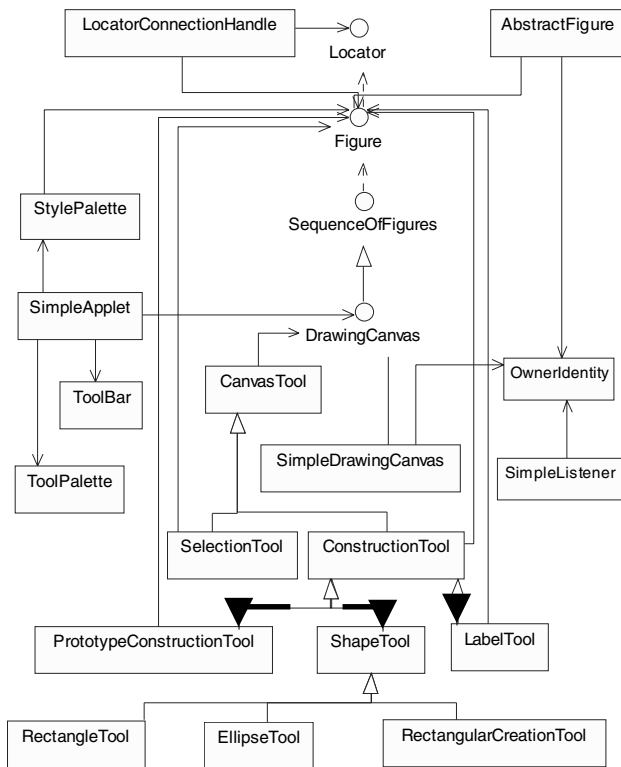


Figure 10: Visiting classes that interact with class `ConstructionTool`

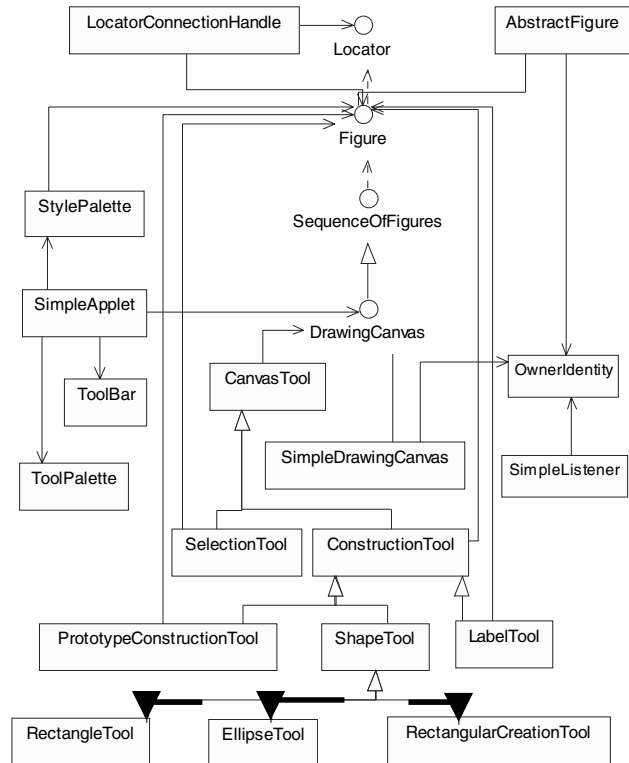


Figure 11: Visiting classes that interact with class `ShapeTool`

Step 4: Visit classes interacting with `ShapeTool`, Figure 11.

Classes `RectangleTool`, `EllipseTool`, and `RectangularCreationTool` call `getCurrentSessionOwner()` of class `ConstructionTool` and `move(...)` function of the `Figure` interface, hence modification is required in all these classes. Their neighbors did not need any modifications and the change did not propagate further.

This completed the change propagation and the code is now consistent again.