Using Clone Detection to Manage a Product Line

Ira D. Baxter
Semantic Designs, Inc.
www.semdesigns.com
idbaxter@semdesigns.com

Dale Churchett
Salion, Inc.
www.salion.com
Dale.Churchett@salion.com

1 Clone Detection to find Domain Concepts

Clone detection finds code in large software systems that has been replicated and modified by hand. Remarkably, clone detection works because people copy conceptually identifiable blocks of code, and make only a few changes, which means the same syntax is detectably repeated. Each identified clone thus indicates the presence of a useful problem domain concept, and simultaneously provides an example implementation. Differences between the copies identify parameters or points of variation. Clones can thus enhance a product line development in a number of ways: removal of redundant code, lowering maintenance costs, identification of domain concepts for use in the present system or the next, and identification of parameterized reusable implementations. A slightly surprising property is that clones sometimes reveal code bugs directly by inspection of parameter bindings with inconsistent actual or conceptual types.

This position paper sketches a work just started, to review the clones found in a Java-based 250K SLOC web application and determine their impact on the current product and its next generation, currently being planned.

2 Application to Salion's product suite

Salion, Inc. provides a product suite solution to its customers comprised of multiple products built from a core component base. The set of products enabled for a customer is determined by Salion Professional Services based on the customer requirements and matched against the capabilities provided by each product within the suite.

The framework created to support the licensing requirements defined by Salion's business model provides seamless integration between each product. Adding new products becomes a composition task for development rather than new development. To enable the level of reuse required for a product suite approach requires a core set of components be developed, tied together with a software configuration management system that handles the legal combinations of core components.

Managing a set of core components is not an easy task and becomes harder the large the system becomes. The aggressive schedules demanded by software consumers and a fast paced development cycle compounds the problem as developers struggle to accomplish their tasks as quickly as possible. In many cases, designing for reuse is the last thing on a developers mind and cut-and-paste programming may win the day.

By applying clone detection as part of a never-ending mining and refactoring operation, Salion hopes to mitigate the risk of cut-and-paste programming and reveal abstractions that are either missing from the component base or to identify services that are not being provided by core components or subsystems that should be.

Early experiences have already proved effective, with some surprising side effects such as detecting bugs in the code, and revealing limitations of dependent technology and for identifying potential future problems with class explosion.

3 Clone Detection on Java

Semantic Designs has been developing automated clone detection and removal technology ("CloneDR") based on comparing sequences of syntax trees [2]. The detection technology is in turn built on top of DMS [1], (www.semdesigns.com/Products/DMS/DMSToolkit) a program transformation system parameterized by a description of the language to be processed. For the purposes of this experiment, a definition of Java was provided to DMS. DMS in turn is based on parallel computing foundations [3] to bring sufficient computing power to bear for the arbitrary analysis and modification problem problems DMS is generally expected to solve. For clone detection, this alleviates the essentially N^2

computation cost induced by comparing every pair of trees.

The Java CloneDR has been applied to two previous large Java software systems, the Sun Swing toolkit (some 230K SLOC having about 10% cloned code by volume, as instances of about 1800 detected abstractions), and a 2 million-line enterprise resource planning system (showing about 12% cloned code by volume, having some 7400 detected abstractions). *It is clear that the clone detector finds large numbers of code fragments deemed by the engineers of being conceptually coherent and useful enough to repeatedly steal.*

We applied the Java CloneDR to Salion's source code base of 257K SLOC. It found 12.5% cloned code as instances of some 1600 detected abstractions. This is remarkably consistent with the other systems. There are about 6500 very small clines, of size 3 lines or less, which may or may not be interesting. At the other end of the spectrum, there are 32 clones of at least 100 lines each in size.

We have plans to apply the CloneDR to Salion's application several times in the coming year, to get a better understanding of how the clone base evolves over time in response to aggressive clone remediation action by Salion.

2 Detecting bugs in clones

Examining the clones for abstractions often leads to bug discovery. Sometimes the bugs are directly obvious on inspection. The following illustrates a clone having 73 instances that fails to set a value to a reasonable result if a conversion error occurs.

Clone 46: 4 parameters, 4 lines from Line 387 to 390
File:…SearchParams.java:

```
try {date = Integer.parseInt(dateStr); }
catch (Exception e) {}
```

Showing parameters as `[[#n]]`, the abstraction for the clone is:

```
try { [[#1]]= [[#2]]. [[#3]]( [#4]]);}
catch (Exception e) {}
```

The cure is replace this with a conversion procedure that complains and produces a useful default value:

```
date = Integer.safeCconvert(dateStr,0);
```

This type of fix is the classic purpose of the CloneDR tool: to help find and repair a mistaken concept repeated many times.

One can discover errors by type mismatches in the

Clone 33: 2 lines from Line 188 to 189
File ….SalionObjectImpl.java

```
public void setCreatorGUID(String creatorGUID)
  { this.creatorGUID = guid;}
```

parameters. This clone:

is a broken setter, and it was discovered because the abstraction had 4 parameters instead of 3. Parameter #4

```
public void [[#1]](String [[#2]])
  { this. [[#3]]= [[#4]]; }
```

should have always been parameter #2:

3 Acting on abstractions found

Some initial analysis of discovered abstractions have already caused architectural change in the next generation product.

The most massively replicated clone is one of the smallest: 1450 instances of the 3-line `get`*`XXX`*`Label()` clone. Discovery of this idiom has the UI developers

```
public String getProfileLabel() {
 return (getString("profileLabel", "Profile"));
}
```

considering the use of a JSP Tag Library instead of putting those methods in the beans. Implementing this would be expected to be a big win because of the way the label mechanism is currently implemented. There are 4 layers of properties files in the application system. The first layer is global information, the 2nd more specific to the UI containing considerable information currently unused. Therefore these procedures are dead and can be removed. Keeping 1500 of these in sync would be a nightmare, where the Tag Library can handle it automatically.

One of the detected clones indicated a problem waiting to happen. The Salion product suite includes many large business objects with many attributes that customers may or may not wish to use. Salion can configure the system to use/not use a subset. of these attributes per customer. To date, only 4 business objects have been enabled with this functionality, but others will follow. . The clone detection results showed an area in this subsystem that would have exploded for every business object added in the future.

Clones were discovered that revealed a mechanism the UI developers created that should have been provided by the core configuration subsystem i.e., the subsystem was not enabling one of their requirements. The mechanism developed by the UI team would have caused an explosion of classes and methods (and clones) had it not been noticed. The service will be refactored into the configuration subsystem in a future release.

The architect is finding it easier to get a handle on the subsystem design using the clone detection results. Going

through each source file is not a good way, nor is reverse-engineering in Rose, tried previously.

4 Next Steps

The clone analysis results are currently being reviewed in detail by the project architect to get a sense of the abstractions discovered, their potential value, and how that value can be realized by either explicit refactoring of the system to instantiate the abstraction directly, or by other modifications of the system to enhance its quality.

References

[1] I. Baxter and C. Pidgeon, "Software Change Through Design Maintenance". *Proc. International Conference on Software Maintenance*, IEEE, 1997.

[2] I. Baxter, et. al., "Clone Detection Using Abstract Syntax Trees", Proc. *International Conference on Software Maintenance*, IEEE, 1998.

[3] *PARLANSE Reference Manual*, Semantic Designs, 1998.