Comp462 : Review Essay

A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint

Daniel Ballinger
Victoria University of Wellington
Wellington, New Zealand
db@mcs.vuw.ac.nz

ABSTRACT

This essay is a critical review of a paper titled "A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint" published by Michele Lanza and Stéphane Ducasse in OOPSLA 2001.

In reviewing the paper I covered the contributions made by the paper, how it advanced software visualisation for reverse engineering over previous papers, and finally the opportunities it raises for future work.

1. INTRODUCTION

First impressions last. If the source code is the first contact a developer makes with a class the impressions they take away are not always favourable. What's needed is a third party to give an introduction that hides away most of the complexities of the classes under inspection and to provide an organisational overview. The methodology presented in the paper lays the foundation for developers in the construction of a mental model that will aid them in the reverse engineering process.

At the centre of this methodology is what Lanza and Ducasse call the class blueprint. This blueprint helps facilitate reverse engineering, which is becoming more important in the computing industry as a technique to combat with the increasing number of legacy systems [MUL 00].

2. OVERVIEW OF THE PAPER

The abstractions that classes provide are vital building blocks in object-oriented design. To help reverse engineers quickly grasp the purpose of a class and its inner structure Lanza and Ducasse propose a visualisation called the Class Blueprint. A class blueprint template, shown in Figure 1, has the components of each concrete class mapped into its 5 layers: initialisation, interface, implementation, accessor and attributes. Only the static structure of the class is considered as to give what Lanza and Ducasse describe as "a ''taste'' of the class: an intuitive and quick understanding of its internal structure and the way it interacts with its super- and sub classes."

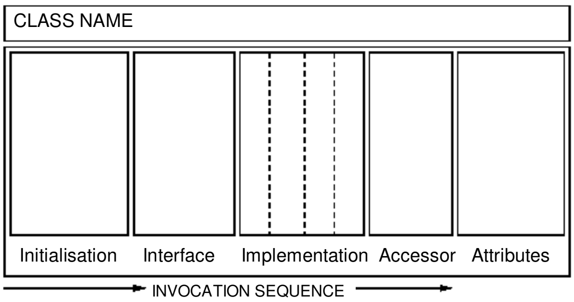

Figure 1: The decomposition of a class into layers. Image from [LAN 01a].

Within the layers of each class, methods and attributes are represent using coloured boxes (sometimes referred to as nodes) of various size and shape. The width and height of the box are assigned to metrics while colour is used for supplementary information such as: is the box an attribute, an abstract method or an extending method. Methods are placed left to right within the layers according to their invocation sequence and context. Coloured lines depict the relations between the boxes shown in the class.

Inheritance relationships between ancestors and descendants of classes are also considered using a layout similar to Figure 2. This allows the distributed nature of a class, like the use of attributes defined in the superclass or the overriding of methods, to be viewed.

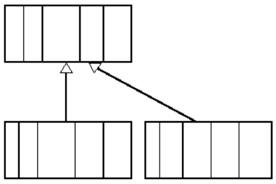

Figure 2: The visualisation of class blueprints in the context of inheritance. Image from [LAN 01a].

Using the experienced gained in applying their approach in several case studies, the authors have proposed a categorisation system for blueprints. Class blueprints are assigned to categories by observing their structure and layout in the visualisations.

3. CONTRIBUTIONS MADE BY THE PAPER

As pointed out in the paper, the main contributions made are the class blueprint for the visualisation of classes and a novel categorisation of classes based on their blueprints.

3.1 The Class Blueprint

The main benefit of the class blueprint is a reduction in complexity early on in the reverse engineering process. The ability to quickly get a "taste" of a class without first reading the source code allows the developer to gain a "quick and intuitive understanding" of the internal representation of the class and how it is structured, and then read the code with a preconception of the design.

When looking at several blueprints along with their inheritance relationships it is possible to observe how a class is embedded in an inheritance hierarchy and how it makes use of inheritance. The incremental definition of classes from superclass to subclass becomes more apparent when viewed from the perspective of the class blueprint. For instance, the developer could see what attributes are used from the superclass and what new methods are introduced.

3.2 Categories

Lanza and Ducasse believe the main strength of the technique they introduce in the paper "is the ability to categorise class blueprints". Some of the main categories introduced in the paper are reproduced in the appendix for reference. The benefits introduced by categorisations are twofold.

Firstly, by understanding what category the class belongs to it is possible to infer properties about it. This can include what the general purpose of the class is and what sort of behaviour is to be expected from it.

Secondly, these categorisations provide a common vocabulary for communication among developers. Lanza and Ducasse point out that this is important "where complex contexts and situations must be communicated to another person in an efficient way."

3.3 Identification of design anomalies

Observing class blueprints either individually or within an inheritance hierarchy can reveal aspects of the design that may need closer inspection. This can range from simple things like attributes and methods that never get used to more complex issues like the misuse of the inheritance hierarchy.

Some examples include:
Inconsistent Accessor Use. Here, as shown in figure 3, the attributes of a class are accessed both directly and by the provided accessor methods. This could indicate a violation of the encapsulation of the class and that an external class may have an unnecessary dependency on the internal structure of the class. Lanza and Ducasse observe that "If the accessor is never invoked then it just adds unnecessary complexity to the class."

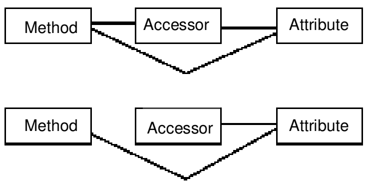

Figure 3: Two cases of inconsistent accessor use. Image from [LAN 01a].

Misuse of Inheritance Hierarchy Policy. Lanza and Ducasse comment that they often find the whole hierarchy is built on one main inheritance concept. Problems can occur if a maintenance programmer adds something to the design without being aware of the main inheritance method used in the design. In such a case, the visualisation and categorisation of the classes with inheritance relationships via class blueprints could help identify the policy in use and hence reduce the possibility of developers introducing inconsistencies into the design.

Classes without a Blueprint. Experiments undertaken in the paper to validate the approach found some classes, often having more than 50 methods, which could not be classified. Lanza and Ducasse observed that "Although such classes may not necessarily be a sign of bad design, we think that the classes are suspicious nonetheless and should be further examined."

3.3.1 Refactoring

Several categorisations of classes were identified where refactoring techniques presented and discussed by Martin Fowler [FOW 99] were applicable.
These include:

Siamese Twins. Lanza and Ducasse found cases where sibling classes had "an impressive similarity with each other in terms of methods, attributes, method invocations and attribute accesses." They observe that in such cases the common functionality could be refactored into the superclass of the siamese twins.

Splittable classes. Splittable classes are those with two (or more) separate clusters of methods and attributes which are not connected in any way. The "Extract class" refactoring suggested by Martin Fowler could be used to create new classes, each with stronger internal cohesion.

Pure Mute Overriders. In this situation, a subclass B does only method overrides of its superclass A without invoking or accessing any of the functionality of A. Refactoring is suggested to move the overridden methods into a new superclass C NEW and make A and B siblings. This is similar to Martin Fowler's "Extract Superclass" refactoring As shown in Figure 4.

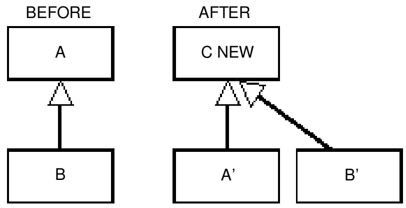

Figure 4: The "Extract Superclass" refactoring. Image from [LAN 01a].

Lanza and Ducasse observe that applying the refactorings can enhance the flexibility and design of the system while helping to prevent code bloat.

3.4 Patterns

In some sections of the paper Lanza and Ducasse mention being able to identify patterns from [GAM 95] using the class blueprints - "… the code contained many pattern-like structures like facades, wrappers, etc." Just as the categorisations they have identified enable the developers to assume things about the design, recognising patterns brings a powerful world of comprehension about the design, especially if the developer has a good understanding of the pattern. Patterns can make the purpose and usage of several classes more apparent, speeding up the early stages of the reverse engineering process.

If collaboration is added to the blueprint model perhaps more patterns could be identified.

4. HOW IT ADVANCED THE STATE OF THE ART OVER PREVIOUS PAPERS

In [CHI 90] Chikofsky and Cross present a taxonomy for terms used in the reverse engineering and design recovery fields. According to the definitions they give, the blueprint visualisation process is targeted mostly at reverse engineering.

4.1 Refactoring

In applying their technique experimentally, Lanza and Ducasse note that the insights into the system gained through the blueprint often trigger refactorings (like those in section 3.3.1), leading to forward engineering and reengineering.

4.2 Layers of software

The diagrams produced in the paper utilise and demonstrate the usefulness of several tools Lanza and others have worked on.

CodeCrawler [DEM 99, DUC 01, LAN 99], a visualisation tool that combines visualisation techniques with metrics and generates the class blueprints presented in the paper. CodeCrawler was written in Smalltalk by Lanza and is utilises Moose.

Moose [DUC 00] is a tool environment to reverse engineer and reengineer object-oriented systems. It consists of a repository to store models of software systems, and provides query and navigation facilities. The models consist of entities representing software artefacts such as classes and methods. Moose is also developed in Smalltalk and is an implementation of the FAMIX metamodel.

The **FAMIX** model [DEM 01] provides a language-independent representation of object-oriented source code and is used by the FAMOOS (Framework based Approach for Mastering Object Oriented Systems) tool prototypes as a basis for exchanging information about object-oriented software systems.

4.3 Viewing the internals of classes

The class blueprint offers a greater depth of detail for the static structure of classes than many of the existing software visualisation tools by treating methods and attributes as the smallest units of visualisation.

Some examples given in the paper of visualisation tools that provide large-scale static views of class structure include Rigi [STO 97] and SeeSoft [EIC 92].

In SeeSoft entire files and the metrics (referred to as statistics in the paper) for the lines of code within them form the units of visualisation. So, while changes to the code are tracked, the internal structure of the class is not made apparent.

Rigi provides two contrasting visualisations of the software structure via graphs. It is mainly targeted at "browsing software subsystem hierarchies."

In [LAN 02] Lanza and Ducasse apply similar techniques to those in the paper under review except the unit of interest is the class.

Lanza and Ducasse observe that while most tools (like those mentioned above) treat the class as the smallest unit of visualisation, some do delve into the internal workings of classes. Those that do visualise the internal workings, like the FIELD programming environment [REI 90] given as an example, do not probe as deeply by limiting themselves to showing "the method names, attributes, etc."

4.4 Visualisation and metrics hybrid

In October 1999 Michele Lanza published his masters thesis titled "Combining metrics and graphs for object oriented reverse engineering" at the University of Bern, Switzerland [LAN 99]. In it he developed a lightweight approach to software reverse engineering by combining simple graphs with simple object oriented metrics. The approach introduced the concept of a node, as seen in figure 5, which allows for the easy visualisation of metrics within diagrams. In a later work [DEM 99], Lanza made refinements to some aspects of the technique, including the layout algorithm.

The techniques Lanza used in his thesis employed the class as the unit of visualisation. Metrics were used to assess the size and internal complexity of the classes and for their scalability. The paper in review also uses metrics to examine the internal workings but advances the process by encapsulating them within the blueprint template that in turn provides a combined view of a class and its internal structure.

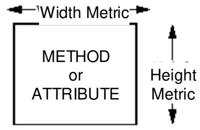

Figure 5: A graphical representation of methods and attributes using metrics. Taken from [LAN 01a].

In *Reverse Engineering: A Roadmap* [MUL 00. Pg 55], the authors observe that by combining different reverse engineering techniques the weaknesses of one can be traded off against the strengths of the other.

> "Each of the investigative techniques just described has certain advantages and disadvantages. However, combining theses techniques … should produce stronger results."

5. OPPORTUNITIES IT RAISES FOR FUTURE WORK

Lanza and Ducasse identify several areas that currently limit their work and could be improved on in the paper.

5.1 Cognitive science

The visualisation algorithm and methodology proposed in the paper is ad hoc and not based on research in the field of cognitive science. They wish to research further into this field and update their approach as required.

5.2 Collaboration between classes

Aspects of collaboration between classes are not currently part of the class blueprint methodology. In the future they hope to extend the methodology to classes that are not within the same inheritance structure but collaborate with each other, presenting an opportunity to identify new classes of categories.

In adding this new information to the class blueprint model it is important that the simplicity of the visualisations is not compromised. Putting too much information into the diagrams could make it harder for the developers to quickly understand the classes. This ties into researching into cognitive science and what can be accomplished with simple visualisations.

5.3 Non-static information

The current class blueprints actually give a visualisation of every possible combination of method invocations. Runtime information could be taken into account to deal with methods that never actually get invoked and attributes that never actually get accessed. This in turn could lead to a greater understanding of how the class actually works and presents an opportunity to identify new categorisations based on dynamic behaviour.

5.4 Layout Algorithm

Lanza and Ducasse point out that some manual post processing of visualisations is still required and that enhancement of the layout algorithm too improve space utilisation could aid readability. They observe that sizeable classes would benefit most from this, as currently it is only possible to clearly identify their monolithic properties. Splittable classes could also benefit from an algorithm that clearly displays the internal partition(s).

5.5 Languages other than Smalltalk

The experiments done within the paper have all used Smalltalk programs, but the metamodel (from FAMIX) is language-independent and provides the opportunity for more diverse trials.

The first opportunity offered by testing the process on other popular languages, such as C++ and Java, is to test if the categorisations are still applicable. Lanza and Ducasse also want to find out if the public interface layer of the blueprint is still valid.

Specialised aspects of languages may lead to the identification of new categories. For example, C++ will have class blueprints specific to multiple inheritance. C++ could also introduce possible problems being an impure object-oriented language, like the treatment of global variables.

5.6 Testing

When the paper was published the authors were the only developers to have carried out the tests of the blueprint method. To assess the effectiveness of the process Lanza and Ducasse want to perform empirical usability analysis and qualitative validation with actual reverse engineers using the system. They wish to compare the efficiency of two groups, one with tool one without.

5.7 Categorisations

In addition to the possibilities for finding new categorisations mentioned above, Lanza and Ducasse want to do more experiments to refine the blueprint-naming scheme.

One of these experiments involves apply the technique to legacy systems to evaluate the percentage of classes that cannot be categorised. They see this as a possible limit to the approach, where it is quite common to find around 20 average sized classes without a clear blueprint.

5.8 Interactivity

The visualisations shown in the paper don't appear to show the names of attributes, methods, or classes. I believe in order to be useful, in aiding developer understanding of classes, there must be some way of finding these names when interacting with the diagrams. This may already by possible but it is not apparent or mentioned in the paper.

5.9 Metrics and Methodologies

There is an opportunity to try using different metrics than those used in the paper to alter the information provided in the class blueprint.

Also, Lanza and Ducasse would like to incorporate the class blueprint technique into their larger methodology for reverse engineering of object-oriented systems [DUC 01].

6. CONCLUSIONS

6.1 Contributions

The class blueprint visualisation creates a unique tool that combines some of the benefits of both metrics and visualisations. The ability to observe a class individually or within an inheritance hierarchy allows programmers to get what Lanza and Ducasse describe as a "taste" of a class. Armed with this information, assumptions can be made about the structure and internal representation of a class. When the inheritance perspective is added, the way a class is embedded into the hierarchy can also be considered.

The real strength of the paper comes from the ability to assign most class blueprints to categories. The categories create a common vocabulary for reverse engineers that can be used to discuss " complex contexts and situations". When patterns are identified from the visualisations the benefits are greater still.

Another contribution is the ability to identify, and possibly correct, design anomalies in the design via the class blueprints. In some cases it is possible to apply design refactorings presented by Martin Fowler in [FOW 99].

6.2 Advancements

This paper represents Michele Lanza's and Stéphane Ducasse's continuing work towards a larger methodology for the reverse engineering of object-oriented systems [DUC 01]. In doing this they have leveraged several projects that they have participated on in the past.

Metrics are used to assess the internal complexity of the classes and then used within the blueprint template. This hybrid provides a combined view of a class and its internal structure in one effective visualisation. This is fairly unique to as most other visualisation tools do not offer the same depth of information.

6.3 Future Opportunities

The built-in support from the underlying layers of software for object-oriented languages other than Smalltalk presents the opportunity to verify current and identify new categories of class blueprints.

In a similar way, adding collaboration relationships between classes and dynamic runtime behaviour to the visualisation model could also identify new categories. In addition to this, design patterns and other useful information could also be extracted.

Testing by reverse engineering teams, both with and without the tool, could help verify the effectiveness of the tool.

7. APPENDIX

7.1 References

[CHI 90] E. J. Chikofsky and J. H. Cross, II. *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, pages 13-17, Jan. 1990.
http://www.infosys.tuwien.ac.at/Teaching/Courses/SWV/papers/chikofsky90.pdf

[DEM 99] S. Demeyer, S. Ducasse and M. Lanza. *A Hybrid Reverse Engineering Approach Combining Metrics and Program Visualization.* Published in the WCRE 1999 Proceedings, pp. 175 - 186, IEEE, 1999.
http://iamwww.unibe.ch/%7Elanza/Publications/PDF/wcre99.pdf

[DEM 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 – the FAMOOS information exchange model.* Technical report, University of Berne, 2001.
http://iamwww.unibe.ch/~scg/archive/famoos/publications.pdf

[DUC 01] S. Ducasse and M. Lanza. *Towards a methodology for the understanding of object-oriented systems.* Technique et Science Informatique, 2001.
http://citeseer.nj.nec.com/ducasse01towards.html

[DUC 00] S. Ducasse, M. Lanza, and S. Tichelaar. *MOOSE: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems.* Software Composition Group, University of Berne. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000.
http://iamwww.unibe.ch/~lanza/Publications/PDF/coset00.pdf

[EIC 92] S. G. Eick, J. L. Steffen, and E. E. S. Jr. *SeeSoft – A Tool for Visualizing Line Oriented Software Statistics.* IEEE Transactions on Software Engineering, 18(11):957-967, November 1992.
http://infovis.cs.vt.edu/cs5984/papers/seesoft.pdf

[FOW 99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[GAM 95] E. Gamma, R.Helm, R. Johnson, and J. Vissides. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.

[LAN 99] M. Lanza. *Combining Metrics and Graphs for Object Oriented Reverse Engineering.*
Master Thesis, University of Bern, Switzerland, 1999.
http://www.iam.unibe.ch/%7Elanza/cgi-bin/thesis.cgi?master.pdf
http://www.mcs.vuw.ac.nz/~db/misc/lanza-m-diploma.pdf
Note: At the time of writing the first URL was not functioning correctly. After email correspondence with Michele, he was kind enough to email a copy to myself, which is available at the second URL.

[LAN 01a] M. Lanza and S. Ducasse. *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint.* Published as Proceedings in OOPSLA 2001 pp. 300 - 311, ACM, 2001. 1-58113-335-9/01/10
http://iamwww.unibe.ch/~ducasse/WebPages/Publications.html
http://iamwww.unibe.ch/%7Elanza/Publications/PDF/oopsla2001.pdf
http://citeseer.nj.nec.com/457299.html

[LAN 01b] M. Lanza and S. Ducasse. *The Class Blueprint - A Visualization of the Internal Structure of Classes.* OOPSLA 2001, Software Visualization Workshop. 2001.
http://iamwww.unibe.ch/%7Elanza/Publications/PDF/sv-oopsla2001.pdf

[LAN 02] M. Lanza and S. Ducasse. *Understanding Software Evolution using a Combination of Software Visualization and Software Metrics.*Published in the LMO 2002 Proceedings (Languages at Modeles a Objets), pp. 135 - 149. Hermes Publications, 2002.
http://iamwww.unibe.ch/%7Elanza/Publications/PDF/lmo2002.pdf

[MUL 00] H. A. Müller, J. Jahnke, D. Smith, M. A. Storey, S. Tilley, and K. Wong. *Reverse Engineering: A Roadmap.* Future of Software Engineering Limerick Ireland, 2000.

[REI 90] S. P. Reiss. *Interacting with the field environment. Software – Practice and Experience,* 20:89-155, 1990.
http://www.informatik.uni-trier.de/~ley/db/journals/spe/spe20.html#Reiss90

[STO 97] A. D. Storey, K. Wong, and H. A. Müller. *Rigi: A Visualization Environment for Reverse Engineering.* ICSE 1997 Boston MA USA. 1997.
http://www.rigi.csc.uvic.ca/

7.2 Glossary

Forward engineering:
[CHI 90] – "… the traditional process of moving from high-levels abstractions and logical, implementation-independent designs to the physical implementation of a system."

Reverse engineering:
[CHI 90] - "… to gain a sufficient design-level understanding to aid maintenance, strengthen enhancement, or support replacement."
- "Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction."

[STO 97] - "the process of analysing an existing system to identify its components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction."

Reengineering:
[CHI 90] – "Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. It generally includes some form of reverse engineering (to achieve a more abstract definition) followed by some form of forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system."

Metamodel:
[http://c2.com/cgi/wiki?MetaModel] -
"A MetaModel is a model that describes a model -- the class variables of a particular kind of model or a way of discovering a model." - RaySchneider

7.3 Categorisations of classes based on class blueprints

The following are the main categorisations introduced in the paper [LAN 01a] and are reproduced here for reference.

Single entry. A class "which has very few or only one entry point to the interface layer. It then has a large implementation layer with several levels of invocation relationships."
Data Storage. A class "which mainly contains attributes whose values can be read and written by using accessor methods. Such a class does not implement any complex behaviour, but merely stores and retrieves data for other classes."
Wide interface. A class "that offers many entry points to its functionality in respect to its overall implementation layer." A data storage class can also be a Wide Interface.
Large Implementation. A class that "has a large implementation layer with many methods and many invocations between those methods."
Delegator. A class "which defines delegating methods. If the class defines only delegating methods, we name it a pure delegator."
Constant Definer. A class "which defines default methods which return constant values."
Small Class. A class "which contains few methods and attributes (if at all). To understand a class it is often enough to know its name, especially if the class is a standalone class, …"

With Inheritance Perspective
Definers. Classes "which reside at the top of a hierarchy. They may define some kind of interface behaviour for their subclasses, apart from providing functionality of their own."
Pluggable. A class "which establishes an inheritance policy by defining abstract methods which must be overridden by its subclasses to make them compliant to the policy."
Attribute Definer. A class "whose purpose it to define attributes (instance variables) which are then inherited, used and accessed by its subclasses."
Specialisers. "leaf classes in inheritance hierarchies and implement and refine behaviour at the bottom of the hierarchies."
Extending. "The subclass contains at least one method which contains a super call."
Overriding. "The subclass contains at least one method which overrides the definition of a method with the same signature in the superclass, i.e., the functionality of the method defined in the superclass is completely redefined."
Adding. A subclass which adds "its own functionality to the one defined higher up the hierarchy by adding methods which are not present in the superclass."
Inbetweeners. Classes which are neither definers nor specialisers. They often show similarities with definers or specialisers corresponding to their proximity with them.

Talking. "The subclass communicates (talks) with its superclasses by invoking that methods and by accessing the attributes of the superclasses."
Mute. "The subclass is ''mute'' if it does neither invoke superclass methods nor access superclass attributes."