The Journal of Systems and Software 88 (2014) 231–249

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Visualizing protected variations in evolving software designs

Sébastien Rufiange*, Christopher P. Fuhrman

École de technologie supérieure, 1100 rue Notre-Dame Ouest, Montréal, Québec H3C 1K3, Canada

ARTICLE INFO

Article history:
Received 7 April 2013
Received in revised form 20 October 2013
Accepted 22 October 2013
Available online 22 November 2013

Keywords:
Software visualization
Software design
Information hiding
Software evolution

ABSTRACT

Identifying and tracking evolving software structures at a design level is a challenging task. Although there are ways to visualize this information statically, there is a need for methods that help analyzing the evolution of software design elements. In this paper, we present a new visual approach to identify variability zones in software designs and explore how they evolve over time. To verify the usefulness of our approach, we did a user study in which participants had to browse software histories and find visual patterns. Most participants were able to find interesting observations and found our approach intuitive and useful. We present a number of design aspects that were observed by participants and the authors using our IHVis tool on four open-source projects.

1. Introduction

Changes in a software project happen for different reasons. There can be bug-fixes to respect quality attributes such as reliability, security or performance. There can be changes due to the uncertainty that is inherent to early phases of risk-driven iterative projects. There can be additional functional requirements. There could be a need to run the software on a different platform or on a different operating system. An important characteristic of an enduring software design is its ability to handle change over time.

Information hiding (Parnas, 1971) is a core principle in structured and object-oriented design. Designs that apply information hiding aim to hide parts of the software that are likely to change in order to reduce the impact of that potential change on other modules. Information hiding favors designs that have loose coupling to the elements that are potentially unstable.

There are different strategies to achieve information hiding. *Encapsulation* has been defined as "building a capsule, [...] a conceptual barrier around some collection of things" (Wirfs-Brock et al., 1990). Encapsulation implies that a designer explicitly specifies the boundary and what is visible to the outside. Programming environments typically offer mechanisms of *access control* of capsules to specify and enforce what is hidden and what is visible. A simple example is a Java class that has private members with public methods. Access-control encapsulation can also be used at the package level in Java, such that classes in one package are invisible to classes outside of that package.

A related idea is the *open/closed principle* (Meyer, 1988), which suggest that new features in a software should be implemented by extensions (e.g., adding new classes and methods to a sub-class) rather than modifications, to reduce the impacts on client modules that depend on existing features. Similarly, the idea of *protected variations* (Cockburn, 1996; Larman, 2001, 2005) seeks to isolate what is change-prone (or unstable) behind an intentionally stable interface. Polymorphism or composition can then be used to define varying implementations while the clients only access the interface. Larman (2001) mentioned that the open/closed principle and protected variations are essentially equivalent to the more generic and fundamental information hiding principle. McConnell (2004) proposed the *iceberg metaphor* as shown in Fig. 1. In terms of the open/closed principle, a module is said to be "closed to modification" yet "open to extension". These relationships are always with respect to a client (or set of clients) that should not be affected by the extensions. This defines a frame of reference relative to client classes.

In practice, popular object-oriented design patterns (Gamma et al., 1994) strive to make designs more tolerant to changes. Many of these patterns make use of protected variations to protect respective client classes from extensions or modifications to the software. These dimensions of extension are intentional, with structures that use polymorphism (e.g., in Strategy and Observer) or composition (e.g., in Facade, Iterator and Proxy). However, despite the intentional dimensions for extension, these patterns are not explicit in their definitions about what is hidden or visible to the clients. Protected variations implies that clients only use the stable interface; the information hiding principle implies they should not know or see the extension classes. Ideally, a designer could specify this with access control. However, this is impractical in some environments with traditional access control mechanisms.

* Corresponding author. Tel.: +1 5146546100.
E-mail addresses: sebastien@rufiange.com (S. Rufiange), christopher.fuhrman@etsmtl.ca (C.P. Fuhrman).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.10.044

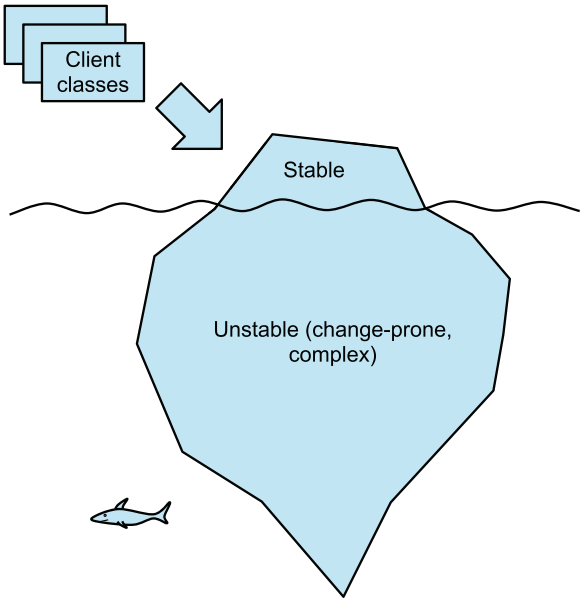

Fig. 1. Iceberg metaphor for information hiding proposed by McConnell (2004). Client classes should not see that which is considered unstable or change-prone.

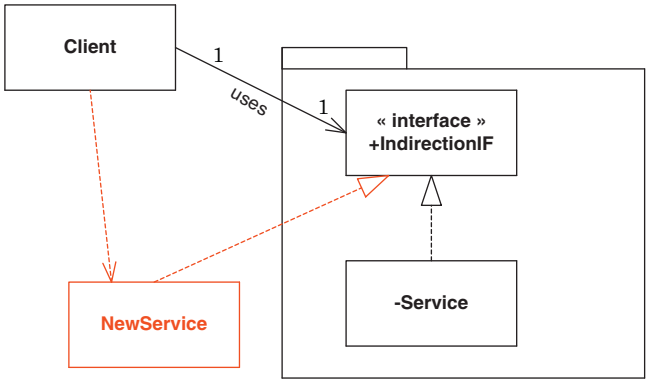

Fig. 2. Interface pattern in Java (Grand, 2002). Package-level access controls prevent coupling to implementations inside the package only.

Extension (hidden) classes could be in various different packages, scattered somewhat arbitrarily throughout a design. It is therefore challenging to determine the boundaries of the *capsule* (i.e., the iceberg).

1.1. Controlling access to unstable elements

Grand (2002) proposed the Interface pattern which is indeed a realization of protected variations. The pattern proposes a solution that keeps "client classes independent of specific data-and-service-providing classes" such that changes to the latter classes will not affect the clients. Fig. 2 illustrates the pattern implemented in the context of a package, such that package access control prevents clients from seeing the implementation (Service) classes. This access control strategy works well provided all the implementations can be constrained within a package. Consider, however, an extension to this design where some NewService is added outside the package. There is no standard way of preventing clients from accessing this NewService directly.

Since strict access control beyond the class level is not commonly used by developers, the principles of "structure hiding" and "information restriction" are not always respected in practice. In fact, the *Law of Demeter* (LoD) essentially says that developers of client classes should be more conservative about using the information they can get from a class (Lieberherr et al., 1988). Also known as "Don't talk to strangers," the LoD could be thought of as follows: every class that a client class accesses *could be* a Facade.[1] By not accessing the "strangers" behind the Facade, a client class is protecting itself from potential variations of those classes.

Because traditional private–public access control mechanisms in languages are often not applied at architectural levels, there exist ad-hoc solutions. One such solution is the "Explicit Extension Rule" in Java Eclipse plug-ins, which is a convention to name hidden, change-prone packages as ".internal." (Gamma and Beck, 2003). The access control to these packages is not enforced by a compiler, but environments such as Eclipse give warnings or errors when code is built[2] if these access control conventions are not followed. Similarly, manifests in OSGi bundles (Hall et al., 2011) can allow defining modular components with access control enforced within the Eclipse Equinox reference implementation.

Some language-specific encapsulation mechanisms exist beyond the class and package level. The "internal" access modifier in C# (Microsoft, 2012) limits visibility of members to files in the same *assembly*, and there is a planned extension to Java to support modules (Reinhold, 2011). However, even with a broader scope of access control, it remains a challenge to assure that the type of extension proposed outside the scope of an arbitrary "module", as depicted by the NewService class in Fig. 2, that is visible to the Client. The problem stems from the fact that modules are specified in a top-down way, rather than being identified dynamically based on coupling.

1.2. Tracking protected variations over time

Differences in the software development process affect the way that designs evolve, and in turn how encapsulation might be specified during the process. As stated by Parnas (1971), it is the designer who decides where the boundary is between what is hidden and what is private. This is traditionally a top-down strategy of design, because important decisions are made *before* client programmers are exposed to the programming interfaces. At the class and package level, things are straightforward: a designer proposes a software class and specifies which parts will be visible to the outside and which parts will be hidden. However, designs involving a lot of composition of reusable libraries are said to be *emergent* (McConnell, 2004). It may not be clear at the higher levels where to apply information hiding; if it is applied too conservatively, it might restrict freedom in bottom-up design. Furthermore, design patterns are often applied in groups (Buschmann et al., 1996). The unstable elements of one pattern might be clients of stable elements of other patterns.

Therefore, specifying access control at the architectural level is arguably more challenging. Again, some solutions exist: the Layers pattern and the Model-View-Controller pattern (Reenskaug, 1979) are both examples of where there is a *convention* of access control. That is, certain lower-level layers (i.e., the Model) should not be coupled to ("see") upper-level layers (i.e., the View).

Keeping track of coupling over the evolution of a project is an important concern for software architects, but so is the tracking of the dimensions of variability. Once a design with protected variations is specified (regardless of access control conventions), several questions remain: (1) does the protected-variations dimension of the design serve a purpose (are new variations implemented)? (2) do the "stable" parts get re-used by more and more clients (are

[1] or the client-facing class of any of the patterns that hide change-prone or complex structures, e.g., Proxy, Iterator, Factory.

[2] http://help.eclipse.org (Plug-in Runtime: Access Rules).