Supporting Ease of Change in the Context of Code

Vibha Sazawal and David Notkin
Department of Computer Science and Engineering
University of Washington, Seattle, WA, USA
Technical Report UW-CSE-2004-09-01

Abstract

Ease of change is an important software property; unfortunately, it is difficult to achieve and sustain. One cause for this difficulty is the semantic gap that divides code from the discourse of ease of change. Design snippets *are partial design representations that scaffold connections between code and ease of change design principles. Design snippets integrate into existing evolution processes by co-displaying selected design information with existing units of code (e.g., files). In this paper, we introduce design snippets and the design principles that inform snippet content. We also present the Design Snippets Tool, which generates four types of design snippets from Java code. Initial assessments suggest that design snippets can be used to identify design problems, plan changes, and confirm that improvements have been made.*

1. Introduction

Software changes [17]; making software easier to change, then, provides value. The prevalence of existing code necessitates ease of change; unfortunately, existing code also complicates attempts to sustain ease of change over the life of a software system. As software engineers modify existing code, they have difficulty reasoning about the effect their modifications have on the overall ease of change of the system. Systems with poor ease of change from the outset – or indeed at any point in time – are difficult to make easier to change.

The ease of change of an existing system can be hard to assess and improve because code is separated from the discourse of ease of change. Consider Parnas' information hiding design rule: Hide each volatile design decision (secret) behind an abstract interface [25]. The terms used in this and other design rules, such as *volatility*, *design decision*, *secret*, and *assumption*, have no direct mapping to code.

We propose that tool support can help software engineers connect code and design principles related to ease of change. In particular, we present *design snippets*: partial design representations that are extracted from source code and intended to scaffold connections between code and ease of change principles. Software engineers can use design snippets as they manipulate code and make decisions related to ease of change during evolution tasks.

Section 2 reviews existing techniques that connect code to ease of change principles and then introduces design snippets. Section 3 outlines the specific design principles that guide our definition of design snippets; in particular, the design snippets presented in this paper support *modular structure*, an important aspect of ease of change. Section 4 describes our prototype tool that generates design snippets. Initial assessment is discussed in Section 5. Sections 6 and 7 present related work and conclusions, respectively.

2. Reducing the gap between code and ease of change principles

Several existing approaches can help software engineers make better decisions by reducing the gap between code and ease of change. Design snippets are intended to complement, not replace, these approaches.

Documentation can be used to explicitly list assumptions and map them to code [3]. This approach relies on the development and maintenance of correct documentation. Unfortunately, many software engineers must evolve systems that lack such documentation.

Another approach codifies solutions to change-related problems in the form of design patterns [8]. Patterns can be immensely useful during restructuring tasks; however, they are not intended to address every change-related problem. In addition, Baniassad and colleagues have shown that a gap exists between design patterns and code that itself must be bridged [1].

Style rules codify ease of change principles in specific language-oriented terms. For example, the Law of Demeter provides guidelines that reduce coupling between objects [20]. Johnson and Foote have introduced other object-

oriented style rules, such as "Separate methods that do not communicate" [12]. Style rules provide useful guidance during implementation and can often be checked by tools. However, style rules are so close to code that they cannot convey the full richness of ease of change principles. For example, no style rule can ensure that all volatile design decisions are hidden, because volatility is dependent in part on requirements.

A new approach: design snippets. To reduce the gap between code and the discourse of ease of change, we propose *design snippets*: abstractions of code that help software engineers reason about specific ease of change principles. Co-displayed with code, design snippets facilitate consideration of ease of change during evolution activities.

To promote compatibility with existing evolution and maintenance practices, design snippets are computed and displayed in the context of *units* of code (e.g., files). Design snippets elide information unrelated to the unit of focus to provide a *partial* but useful representation of ease of change information. Snippets are automatically extracted from code, simplifying their use. They are also lightweight, ensuring that they can be quickly extracted and easily kept up-to-date during viewing and modification of code.

3. Ease of change principles that guide design snippet content

In this paper, we focus on design principles that affect *modular structure*. Parnas defines modular structure as "the decomposition of programs into modules and the assumptions that the team responsible for each module is allowed to make about the other modules" [27, page 2]. Design snippets help software engineers reason about two principles related to modular structure: information hiding and low coupling. We review these two principles below.

Information hiding. The information hiding principle states that "system details that are likely to change independently should be the secrets of separate modules; the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change" [27, page 2]. Every module has an *abstract interface*, which is the set of assumptions that all clients can make [3]; when information hiding is observed, the abstract interface only includes information that is unlikely to change. When change occurs to a module's implementation, the effect of the change is limited to that module.

Information hiding is immensely powerful, but it fundamentally relies on the designer's ability to identify likely future changes. Moreover, even when software engineers properly anticipate changes, they may make errors in design (e.g., unintentional assumptions). The effect of unanticipated changes and errors on a modular system is that interfaces change when implementation details change. When an abstract interface changes, all clients of the abstract interface are potentially affected.

Low coupling. One way to reduce the cost of interface change, as noted by Lieberherr [18], is to reduce the number of clients each module has. This advice can be restated as the well-known design principle of *low coupling* [38]. When coupling between modules is low, the occurrence and effect of interface change is reduced.

If volatile details must be revealed by a module, then a more specific version of the low coupling principle applies. This version states that access to volatile details should be restricted to privileged clients [3]. If this rule is followed, then implementation changes do not affect non-privileged clients because they use a narrower, stabler interface.

4. Design Snippets Tool

The Design Snippets Tool intends to bridge the gap between code and the principles of information hiding and low coupling. Implemented as a plug-in to the Eclipse Java IDE [6], the tool generates design snippets from Java code. Snippets are displayed in a window below the Java code editor. Since Eclipse programmers edit Java files, the "unit" of code associated with each design snippet is a Java file. As files are edited, the tool updates the snippets displayed.

4.1. Mapping between modular structure and code

When designing the tool, we made several decisions that enable extraction of useful design representations from Java source. First, we limited the scope of the tool to Java programs in the Eclipse IDE. Second, we identified common mappings that software engineers make (possibly implicitly) when they define modular structure in Java code. We also identified specific violations of the principles described earlier that can arise as these mappings are used. Design snippet content was then created to help software engineers detect these violations and plan improvements.

From modules to code. The first mapping is that the closest Java construct to a module is a class. To achieve information hiding, Parnas states that designers should first identify volatile assumptions and then design "a module (a collection of subroutines or macros) that 'hides' or contains each one" [26, page 260]. The class, a collection of subroutines with modifiers that restrict access, is a compatible and common Java medium for hiding decisions.

Unfortunately, the mapping from module to class has flaws; for example, Parnas defines a module as a work assignment, but multiple classes often form a single work assignment. The "work assignment" definition of module has two advantages: (1) design decisions become part of the module [25], and (2) software engineers do not have to hide decisions from other parts of their own code [24]. Thus,

the mapping from module to class can cause two violations of modular design principles: (1) design decisions may be made implicitly as classes are defined (and thus not hidden appropriately) and (2) software engineers may fail to hide volatile details from other classes also under their responsibility.

From abstract interfaces to code. The second mapping is that the closest Java construct to an abstract interface is the non-private subset of a class signature. Every class has a signature that describes how clients should access class functionality.[1] Unfortunately, the interface between two modules, as defined by Parnas, is intended to include *all* assumptions made by the modules about each other [24]. The non-private class signature is inherently a subset of all assumptions. As a result, when defining and using class signatures, two violations of modular design principles can occur: (1) non-private class signatures may reveal volatile design decisions and (2) clients may use information not actually revealed by non-private class signatures. These violations are compounded by high coupling between classes.

Additional Java features. Inheritance and Java interfaces are also used to define modular structure in accordance with design principles. For example, one way to shield clients from implementation change is to substitute one class with another class that shares the same superinterface. Two other techniques are replacement of a class with a subclass and replacement of one subclass with another subclass. In cases when those approaches are intended, the spirit of modular design principles can be violated if classes offer or use details not revealed by a superclass or superinterface signature.

Goal of snippet content. The goal of snippet content, given these mappings, is to help software engineers identify violations of modular design principles and plan improvements. In the next four sections, we introduce each snippet and show screenshots generated by our tool. To illustrate the reasoning that the tool supports, we analyze three Java versions of the classic KWIC index example. Parnas presents two modularizations of KWIC: in *Modularization 1*, data representation is shared knowledge; and in *Modularization 2*, modules interact through information-hiding interfaces [25]. We present snippets generated from both versions and *Modularization 3*, a new version that employs inheritance.

4.2. Information hiding snippet

The *information hiding snippet* assists the software engineer in her assessment of the separation between interface and implementation. Software engineers can use the snippet to identify when interfaces reveal volatile information.

Overview. The information hiding snippet presents two outlines side-by-side. The scope of both outlines presented is the active file being edited or viewed in the Eclipse IDE. For each class[2] defined in the file, the *interface outline* describes the non-private class signature and the *implementation outline* describes the class' implementation details. Specifically, the interface outline presents details about non-private field signatures, method signatures, member types, superclasses, and superinterfaces. The implementation outline presents details about private fields, private initializers, and private method signatures. The implementation outline also includes a description of *secret* types used by each method.

Secret types. Secret types are types whose class or instance members are used by a class but whose use is not obvious from perusal of the class signature. The types of parameters to methods and the types of class and instance variables are not secret. Other non-secret types are `this`, an enclosing type, or a nested type. The implementation outline of the information hiding snippet displays the secret types used by each method of each class and a description of how that type is used. For our purposes, a type or type instance is *used* if (1) it is a parameter to a method, (2) if it is a target of a method call, (3) if it is created by a constructor call, (4) if a class variable or instance variable is accessed, or (5) if an instance is caught as an exception.

Secret types are interesting because they provide a succinct view of the implementation details ostensibly hidden by an interface. Many noteworthy implementation details are captured in a list of instances created, exceptions caught, and other uses of non-parameter, non-field types. In addition, secret uses of types are naturally expressed in terms that are easy to compare to the signatures in the interface outline. Together, secret types and private members provide a meaningful view of class implementation details in a reasonable amount of screen real estate.

Example. The following example illustrates how the information hiding snippet can bring attention to violations of information hiding. Figures 1a and 1b show the interface outline and implementation outline for a circular shifter based on Modularization 1. Class `CircularShift` prepares an index of all circular shifts as a two-dimensional array. Each column of the array lists the starting address of the circular shift in memory and the original index of the unshifted line.

Figures 2a and 2b show the interface outline and implementation outline for a circular shifter based on Modularization 2. In this version, clients call methods of

[1] Behavioral interface specification languages, such as JML [16], and design-by-contract tools, such as iContract [15], augment Java with additional constructs for expressing specifications. Because none of these features appear in Java by default, they are outside the scope of this paper.

[2] Java interfaces are also analyzed by the information hiding snippet, but we omit discussion of them here.

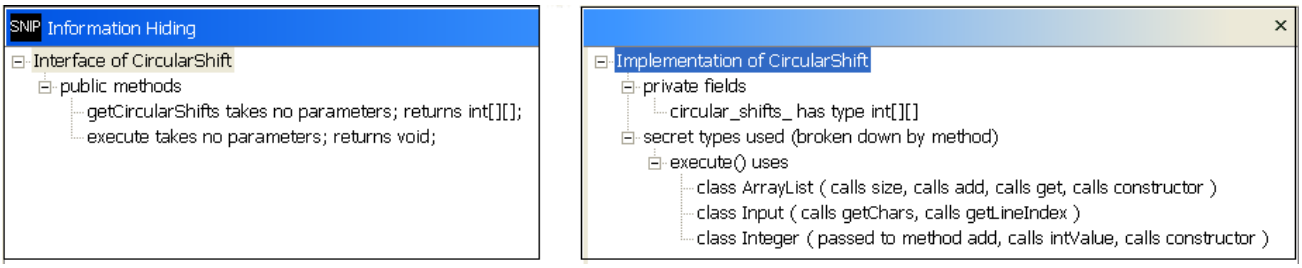

Figure 1. (a) Interface outline (left) and (b) implementation outline (right) for mod. 1's circular shifter

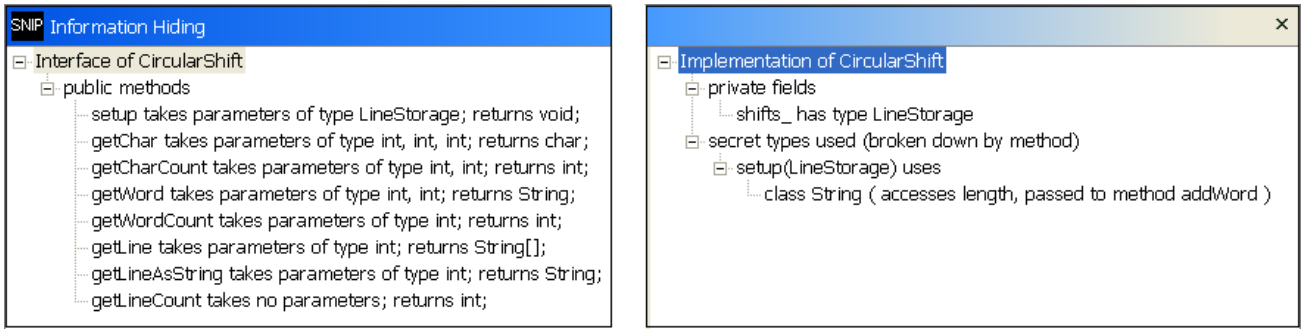

Figure 2. (a) Interface outline (left) and (b) Implementation outline (right) for mod. 2's circular shifter

`CircularShift` to obtain the words or lines comprising each circular shift.

Comparing Figure 1a to 1b, we see that `CircularShift` stores circular shifts as an `int[][]` (field `circular_shifts_`) and offers an `int[][]` to clients (via method `getCircularShifts`). This similarity between field type and return type suggests that the signature of `CircularShift` may be sensitive to changes in the way circular shifts are internally stored. The only access clients have to circular shifts is `getCircularShifts`; thus, all clients are affected if the signature of `getCircularShifts` changes. Moreover, clients may need to change even if `getCircularShifts`'s signature does not change, because clients are also sensitive to changes in the way data within the integer array is organized.

We also note that a secret type used by `execute` is `Input`. `Input` creates an index that contains the starting address of each line. Figure 1b shows that `CircularShift` calls `Input.getLineIndex` to obtain that index. In summary, Figures 1a and 1b reveal that volatile data representations are shared widely by `Input`, `CircularShift`, and `CircularShift`'s clients.

In contrast, Figures 2a and 2b show that Parnas' second modularization hides volatile design decisions more effectively. Figure 2a shows that Modularization 2's `CircularShift` has a method named `setup` which takes a `LineStorage` instance as a parameter. This `LineStorage` instance contains the equivalent of the line index obtained by directly calling `Input.getLineIndex` in Modularization 1. The actual retrieval of the `LineStorage` index (from `Input` or another module) is left to `CircularShift`'s client in this modularization. The remaining methods of `CircularShift` provide clients with means for accessing words and lines of the circular shifts. None of the signatures of these access methods reveal the implementation details shown in Figure 2b. Figure 2b shows that `CircularShift` chooses to store the circular shifts in a `LineStorage` instance, but Figure 2a shows that clients cannot directly access that instance.

Implementation details. The Design Snippets Tool uses Eclipse APIs [6] to access abstract syntax trees and objects associated with each Java file. For performance reasons, the information hiding snippet offers two analysis modes: a quick mode and a slow mode. The modes affect the completeness of the list of secret types. To determine the types of objects used in the active Java file, the quick mode uses local variable declarations, method signatures, and field declarations. The quick mode identifies static member accesses using a list of types compiled from the import statements in the file.[3] The slow mode requests type bindings from the Eclipse compiler to determine the names of types that are used but not named in the active Java file.

One set of identifiers whose types might not be named

[3]The quick mode also obtains a list of all types in the current package.

within a class body is fields of superclasses. This problem highlights a fundamental issue with the mapping of module to class – are superclass members part of the module, or are they distinct from the module? From the perspective of a client of a class, non-private members defined in the superclass are not distinct from members defined in the class itself. But from the perspective of the superclass and subclass, the subclass is a client that is affected by changes to the superclass. As a result, the quick mode lists the superclass as a secret type if explicit calls to superclass methods (i.e., calls preceded by the keyword `super`) are made. The slow mode also lists the types of superclass fields used and the superclasses of enclosing types (if used) as secret types.

4.3. Type assumptions snippet

If the non-private signature of a class hides volatile design details, then clients that only use the non-private signature are protected when details change. However, clients commonly make assumptions that go beyond what class signatures actually reveal. These assumptions, often inadvertent, can result in widespread sharing of volatile details. Similarly, classes can make implicit assumptions about their clients, resulting in adverse effects when clients change. Examples include assumptions about performance, order of operations, and run-time types of signature elements.

Overview. The *type assumptions snippet* helps software engineers detect assumptions made about types specified in a class signature. Type assumptions are manifested when casts are made from the types specified in the class signature to other types. For each type defined in the active Java file, the type assumptions snippet lists casts to parameters and return values of the type's methods and casts made to the type's fields.

We chose to focus on type assumptions because they are straightforward to detect and are often symptoms of larger problems with information sharing. For example, type-casting of return values and parameters may indicate that intended module boundaries are being bypassed and that details known to be volatile are being used. A common type-cast is from superclass to subclass, and these casts reduce the software maintainer's ability to substitute one subclass with another in response to change. The maintainer's problem is compounded by the fact that casts by clients can only be identified by perusal of client code. Software engineers can use the type assumptions snippet to decide whether the benefit of type-casting (if any) outweighs the potential maintenance cost of sharing information not documented in any class signature.

Example. This example illustrates the decision-making support offered by the type assumptions snippet. In this example, we present a new version of KWIC, which we call Modularization 3. In Modularization 3, the input processor and the circular shifter both inherit from `IndexCreator`. `IndexCreator` defines a protected instance variable of type `Index` (named `m_index`) and a public method named `getIndex()`. Two classes inherit from `Index`: `LineIndex` and `ShiftIndex`. Figure 3 shows a subset of Modularization 3's class diagram.

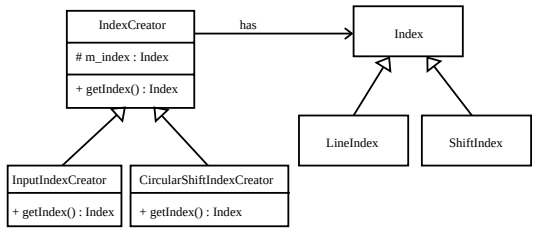

Figure 3. Subset of mod. 3's class diagram.

`InputIndexCreator` instantiates the inherited field `m_index` as a variable of type `InputIndex`. `CircularShiftIndexCreator` instantiates `m_index` as a variable of type `ShiftIndex`. The type assumptions views for `InputIndexCreator.java` and `CircularShiftIndexCreator.java` are shown in Figure 4. Three clients cast the return value of `InputIndexCreator.getIndex()` to `LineIndex`, and one client casts the return value of `CircularShiftIndexCreator.getIndex()` to `ShiftIndex`.

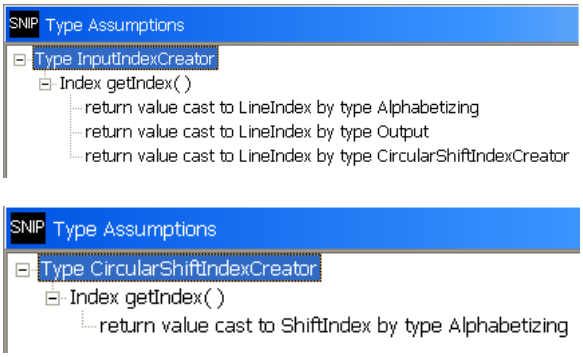

Figure 4. Type assumptions view for mod. 3's input processor and circular shifter

Suppose Alice, a software engineer, decides to replace both `LineIndex` and `ShiftIndex` with a single class called `LineStorageIndex`. (Motivation for this change can be found by reviewing the information hiding snippet views for `LineIndex.java` and `CircularShiftIndex.java`, not shown here.) Without the type assumptions snippet, Alice may dangerously infer that her change will not affect clients, as long as `LineStorageIndex` offers a `getIndex` method that returns an instance of type `Index`. But when using the type

assumptions snippet, Alice will immediately note that the return value of `getIndex` is cast by clients to more specific subclasses.[4] Alice now has a more complete understanding of the impact of her intended change; in particular, she now knows that clients actually rely on specific `Index` subclasses.

Because the type assumptions view displays casts that appear in client code, the software engineer can review a class signature *and* its (mis)use at the same time. Often problems with a class signature can be surfaced by viewing how the signature is used. In the case of Modularization 3 described above, clients make type-casts because they need to access specific methods of `LineIndex` and `ShiftIndex`. The `Index` class signature fails to offer enough functionality to `IndexCreator` subclasses. In this case, the software engineer needs to weigh the benefits of inheritance with the potential cost of general class signatures that meet few client needs. Type assumptions are a tangible way to detect these tensions. In addition, type assumptions have a real effect on the ease of change of a system.

Implementation details. The type assumptions snippet uses information in the active Java file to identify casts to method parameters. To determine if clients cast fields or method return values, the snippet analyzes all files in the active Java project[5] the first time it runs. The snippet caches all casts that are discovered and then displays those casts that are relevant to types defined in the active Java file. The cache is updated as files are edited.

When analyzing a Java file, the snippet looks at variable declarations and assignment statements to identify simple aliases of parameters, return values, or fields. Casts of these aliases are also maintained in the cache. Casts by `this` of superclass fields and return values to methods defined in a superclass are listed in the type assumptions view of the appropriate superclass. Casts by a non-subclass client are assigned to the compile-time type of the client's server, even if the server's method or field was actually defined in a superclass.

4.4. Dependencies snippet

The information hiding and type assumptions snippets help software engineers create a modular structure that isolates volatile details behind interfaces. However, interface change may still occur, and low coupling is recommended as a way to reduce the effects of interface change. The *dependencies snippet* helps the software engineer assess the degree and nature of coupling between classes.

[4] In-package clients can also directly access protected field `m_index`. If any clients cast `m_index` to another type, those casts would also appear in the type assumptions view.

[5] In the Eclipse IDE, a project is a named collection of related Java files.

Overview. For each type *T* defined in the active file, the dependencies snippet displays the types that *T* depends on and the types that depend on *T*. A type can be a class or Java interface; the dependencies snippet treats them similarly. The snippet displays a graph; types are nodes and dependencies are edges. Types with the same qualifiers are grouped together to improve readability, and users can filter uninteresting nodes or request that an entire Java package be represented as a single node. Edge labels describe the nature of the relationship between the two types.

Edges between types are broken down into two categories. Tables 1 and 2 enumerate both categories. The first category consists of edges between T_1 and T_2 where T_2 is statically referenced in T_1's type declaration. The second category consists of edges between T_1 and T_2 where T_1 obtains an instance of T_2 at runtime.

Table 1. Edges between T_1 and T_2 where T_2 is statically referenced in T_1's type declaration

Example	Edge Label
T_1 extends T_2	`extends`
T_1 implements T_2	`impl`
A method of T_1 takes an instance of T_2 as a parameter	`param`
T_1 declares a field of type T_2	`has`
T_1 accesses a static member of T_2	`stat`
T_1 checks if an expression is of type T_2	`inst-of`

Table 2. Edges between T_1 and T_2 where T_1 obtains an instance of T_2 at runtime

Example	Edge Label
T_1 creates a new instance of type T_2	`new`
T_1 obtains an instance of T_2 as a return value of a method	`ret`
T_1 obtains an instance of T_2 as a field of another type	`fld`

T_1 obtains an instance of T_2 by casting another type	`cast`
	`ret-cast`
	`param-cast`
	`fld-cast`

Edge labels. Low coupling is desirable, but some dependencies are inevitable. Two factors to consider when evaluating a dependency include (1) the purpose of the dependency and (2) the dependency's effect on a system's overall vulnerability to interface change. The dependencies snippet produces *edge labels* to support the assessment of these two factors.

Edge labels help diagnose the effect of a dependency by indicating the origin of the coupling between two types. The origin of a dependency affects whether a dependency between two types is obvious from perusal of a class signature or whether the dependency is more obscure. Obviousness affects the ease with which interface changes can be addressed [20]. For example, `param` and `has` dependencies are easier to detect and reason about than many `ret` and `cast` dependencies. In fact, the Law of Demeter outlaws a subset of `ret` dependencies. Edge labels also help programmers estimate the cost of removing a dependency.

Comparison to style rules. Why are style rules such as the Law of Demeter insufficient? We appreciate style rules and encourage their use; however, software engineers possess knowledge about the likelihood of changes that is application-specific and cannot be encoded in any style rule. The dependencies snippet helps software engineers apply their knowledge of volatility to code by displaying a convenient view of dependencies in code of active interest.

Example. Figures 5 and 6 show the dependencies view for `Input.java` in Modularization 1 and Modularization 2 respectively. Dependencies on classes in `java.lang` and `java.io` have been elided.

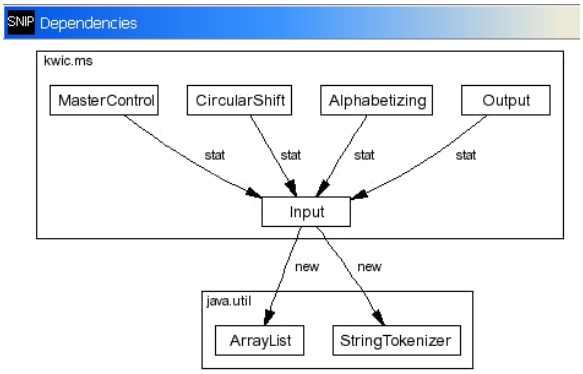

Figure 5. Dependencies view for mod. 1's input processor

When viewing Modularization 1's `Input` class definition, a software engineer can learn from Figure 5 that all other KWIC classes directly access `Input` functionality. If the signature of `Input` changes, all four clients may be affected. The edge label between `Input` and all of its clients is `stat`; since class members are not dynamically bound, the ability to replace `Input` with a subclass is impaired. In contrast, Figure 6 shows that only the `MasterControl` class depends upon Modularization 2's `Input`. Modularization 2's classes are less coupled to the `Input` class and thus more resilient to changes to `Input`.

Implementation details. For each type *T* defined in the active file, the dependencies snippet identifies what types *T* depends on by analyzing the relationships expressed in

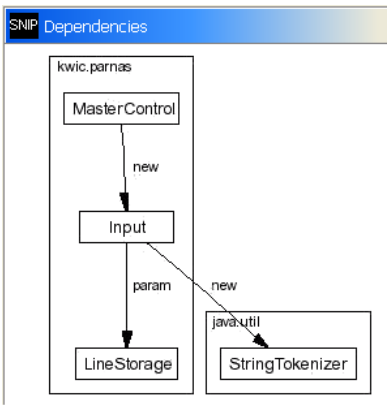

Figure 6. Dependencies view for mod. 2's input processor

that file. To identify the set of types that depend on *T*, the snippet analyzes other files. Similar to the type assumptions snippet, the dependencies snippet analyzes all files in the active Java project during its first run. It then caches these dependencies and displays those relevant to types defined in the active file. The cache is updated as files are edited. Graph layout is performed by AT&T's *graphviz* software package [7].

Similar to the information hiding snippet, the dependencies snippet has a quick mode and a slow mode. The modes affect completeness. The quick mode identifies types whose names appear in the Java file where the dependency is manifested. The slow mode adds `ret` edges when method calls are directly used as the target of other methods or as method parameters. The slow mode also identifies when a type uses fields of its superclasses or the superclasses of an enclosing type. Edges to superclass field types receive the `super-has` label.

4.5. De facto interfaces snippet

While the dependencies snippet provides a quick awareness of dependencies, it does not convey exactly which members are accessed by each client. For all types defined in the active file, the *de facto interfaces snippet* describes the de facto interface of each client. The *de facto interface* is the set of members actually used by a client [14]. The de facto interfaces snippet supports more detailed consideration of the cost of removing a dependency. The snippet also helps software engineers ensure that volatile interface details are restricted to privileged clients.

Example. After looking at the dependencies snippet view for Modularization 1's `Input` class, a software engineer may be interested in learning why so many classes depend on `Input`. This question can be quickly answered by the de facto interfaces view for `Input`, shown in Figure

7. `CircularShift`, `Alphabetizing`, and `Output` all access `Input` member functions to obtain the line index and the character array prepared by `Input`. The de facto interfaces snippet can be configured to organize its results either by client or by member; in Figure 7, the two organizations are shown side-by-side.

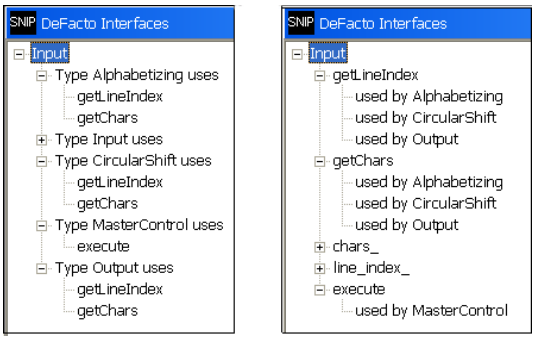

Figure 7. De facto interfaces view for mod. 1's input processor.

5. Assessment

The KWIC examples in the preceding sections convey how design snippets can be used to identify design problems and provide information needed when making decisions related to ease of change. We continue to explore other demonstrations of snippet usage and solicit feedback from volunteer users. In this section we present two examples of snippet usage: an exploratory study in which eight participants used design snippets during a restructuring task, and an example of our own use of design snippets while modifying the Design Snippets Tool.

5.1. Exploratory study

In another paper [31], we describe an investigation into the value of design snippets during decision-making. Eight participants were asked to restructure a small Java application in anticipation of planned changes. The application validated text input and the hypothetical changes involved the addition of new input fields. Participants were introduced to design snippets and were also free to use other features available in the Eclipse IDE. After each session, a narrative of each participant's actions was created by merging notes from two observers with recorded screen capture data.

The narratives suggest that design snippets hold promise. Design snippets were used to discover design problems, identify restructuring goals, plan restructuring activities, examine effects of changes on design, and discover relevant questions about the code. Most participants switched numerous times between design snippets and code, in accordance with our intended usage scenario. Study participants viewed snippets as they made changes to the code and after they had completed making changes. We believe that the scoped display of design snippets facilitated such frequent switching.

Participants appreciated the non-local information provided by design snippets. One participant viewed the casts in the type assumptions snippet and stated "Cool, I can get it over here"; she appreciated that casts of return values of methods can be viewed with the method definition, even though the casts appear elsewhere. Another participant said that the dependencies snippet is "very useful," because it identifies a class' clients when the class definition is active.

5.2. Design snippets as dog food

We actively use the Design Snippets Tool as we develop the tool itself. Design snippets help us *hill-climb* incrementally toward improved ease to change. We present an example of our snippet usage during an enhancement task below.

The enhancement task. Due to user feedback, we added a feature to the de facto interfaces snippet that supports navigation from entries in the snippet's tree control to lines of code. If a user double-clicks on an entry in the tree control, the code related to that entry gains focus. Two classes involved in the enhancement include `DeFactoInterfacesView`, which displays the tree control in the Eclipse IDE, and `DeFactoDoubleClickListener`, a new class that implements double-click support. Double-click support is an unstable feature and many design decisions were tentative. We were interested in reducing the effect of changes to double-click support on `DeFactoInterfacesView`.

Use of dependencies snippet. As we added this new feature, we frequently viewed the dependencies snippet for `DeFactoInterfacesView.java`. We sought to ensure that only one new dependency was added – an edge labeled "has, new" between `DeFactoInterfacesView` and `DeFactoDoubleClickListener`. We did not want `DeFactoInterfacesView` to gain additional dependencies that could result in changes to `DeFactoInterfacesView` if decisions related to double-click support changed.

Use of information hiding snippet. We also viewed the information hiding snippet for `DeFactoDoubleClickListener.java` to obtain a quick view of `DeFactoDoubleClickListener`'s non-private class signature. Ideally, `DeFactoDoubleClickListener` should reveal few implementation details. After viewing the snippet, we decided to change the access level of several auxiliary methods to `private` in order to ensure that clients use the narrowest interface possible.

Use of de facto interfaces snippet. We realized that `DeFactoDoubleClickListener`, as originally envisioned, needs detailed knowledge about tree control entries in order to find the corresponding lines of code. We decided to let `DeFactoDoubleClickListener` call accessor methods associated with each entry in the tree control. The de facto interfaces snippet helps us ensure that only `DeFactoDoubleClickListener` uses these privileged accessor methods.

Discussion. In the future, we may choose to restructure the double-click functionality described above. The use of privileged information by `DeFactoDoubleClickListener` renders it vulnerable to changes if any of the entry classes change. Our purpose here was not to show a perfectly designed feature; rather, we show how design snippets can be used to incrementally reason about an imperfect system and improve it. This *hill-climbing approach* to improved modular structure has been very valuable to us as we write software. Design snippets encourage us to think about modular structure, even when working on features that are small or need to be completed quickly. Snippets also expedite the process of making decisions and monitoring the effects of those decisions.

6. Related Work

Partial views. Slicers and concern-support tools create partial views of a system. Slicers [37, 36] identify lines of code associated with one or two program points. Aspect Browser [9] and FEAT [30] identify cross-cutting concerns in source code. Aspect Browser uses a map metaphor to display where concern code is located. FEAT describes structural relationships between elements in concern code using a tree control. Both Aspect Browser and FEAT require that users inform the tool about what code is initially of interest.

The Design Snippets Tool provides a different partial view. The code of interest is the set of classes defined in the active file. Design snippets integrate into existing evolution processes by augmenting the dominant "editable unit" of code – the file.

Program understanding tools. Program understanding tools help software engineers systematically navigate and explore software systems. Examples include Rigi [21], which visualizes a hierarchical structure of a software system, and SHriMP [33], which presents the structure of a software system as a nested graph.

The Design Snippets Tool complements tools like these. Rather than facilitating navigation and comprehension, design snippets facilitate evaluation of software with regard to certain design principles. Specific support for evaluation is important, because good program understanding tools may obscure design problems; for example, Storey and colleagues noted that users of Rigi and SHriMP failed to notice that the subsystem hierarchy displayed by both tools was not an inherent part of the software [34].

Model-driven development tools. Model-driven development tools, such as Rational Rose XDE [28], Together [2], and Fujaba [22], support the creation of both UML diagrams and code. These tools support automatic generation of code from UML and vice versa. The main difference between these tools and the Design Snippets Tool is the form and role of design representations. Model-driven tool users employ UML to express full descriptions of the static structure and dynamic behavior of software systems. In contrast, design snippets provide partial, targeted information intended primarily for evaluation of software with regard to ease of change.

Design critics. Design critics [29] automatically critique designs. ArgoUML [35] has a set of built-in design rules for UML diagrams. Tools also exist to detect violations of the Law of Demeter [19]. While the Design Snippets Tool also supports specific rules, it does not explicitly identify rule violations. Instead, software engineers use design snippets to manually assess tradeoffs.

Software metrics. Quantitative software metrics can be used to identify design flaws; for example, Chidamber and Kemerer [4] used the "number of children" metric to detect misuse of subclassing in a large user interface class library. Design snippets complement metrics; while metrics can identify that a problem may exist, a software engineer still needs to to confirm, understand, and solve the problem. Software engineers can use design snippets as they investigate possible problems uncovered by metrics.

Advanced paradigms and environments. Aspect-oriented programming [13], subject-oriented programming [10], and functional programming [11] provide features that support ease of change in ways that traditional object-oriented languages do not. For example, an aspect or hyperslice can encapsulate a volatile decision that is cumbersome or impossible to encapsulate in Java. Design snippets for these paradigms appear to require a different mapping from language constructs to design principles terminology. Languages or environments that support specifications [16] or contracts [15] may also require a different mapping. New design principles may also apply.

Advanced environments such as HyperJ [23] and Coven [5] replace the file with other notions of "editable unit." Design snippets should be scoped differently for those environments.

7. Conclusion

Design snippets are abstractions of code that help software engineers improve the modular structure of their software. Compatible with existing software evolution pro-

cesses, design snippets present streamlined design details related to code of interest. Software engineers can use design snippets to hill-climb incrementally from an existing system to an improved one as they perform evolution tasks. Future work includes application of the design snippets concept to additional design principles and programming paradigms.

8. Acknowledgments

The Java implementations of KWIC used in this paper are adapted from a set of KWIC implementations by Nick Scerbakov of the Institute for Information Systems and Computer Media, Austria [32].

References

[1] E. Baniassad, G. Murphy, and C. Schwanniger. Design pattern rationale graphs: Linking design to source. In *Proc. of the 25th Intl. Conf. on Software Engineering*, 2003.

[2] Borland Together. [http://www.borland.com/together].

[3] K. Britton, R. A. Parker, and D. Parnas. A procedure for designing abstract interfaces for device interface modules. In *Proc. of the 5th Intl. Conf. on Software Engineering*, 1981.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. on Software Engineering*, June 1994.

[5] M. C. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *Proc. of the 8th Intl. Symposium on the Foundations of Software Engineering*, 2000.

[6] Eclipse Foundation. Eclipse. [http://www.eclipse.org].

[7] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz. [http://www.research.att.com/ sw/ tools/ graphviz/].

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] W. Griswold, J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of the 23rd Intl. Conf. on Software Engineering*, 2001.

[10] W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proc. of the Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 2002.

[11] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2), 1989.

[12] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June/July 1988.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the European Conf. on Object-Oriented Programming*, 1997.

[14] W. Korman and W. Griswold. Elbereth: Tool support for refactoring Java programs, 1998. Technical report, Univ. of California, San Diego, Dept. of Comp. Sci. and Engineering.

[15] R. Kramer. iContract: the Java design by contract tool. In *Technology of Obj.-Oriented Languages and Systems*, 1998.

[16] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a java modeling language. In *Formal Underpinnings of Java*, 1998.

[17] M. Lehman and L. Belady. *Program Evolution: Processes of software change*. Academic Press, 1985.

[18] K. Lieberherr. Controlling the complexity of software designs. In *Proc. of the 26th Intl. Conf. on Software Engineering*, 2004. Keynote paper.

[19] K. Lieberherr, D. Lorenz, and P. Wu. A case of statically executable advice: Checking the law of demeter with aspectj. In *Proc. of the 3rd Intl. Conf. on Aspect-Oriented Software Development*, 2003.

[20] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In *Proc. of the Object Oriented Programming, Systems, Languages, and Applications Conf.*, 1988.

[21] H. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proc. of the 10th Intl. Conf. on Software Engineering*, 1988.

[22] U. Nickel, J. Niere, J. Wadsack, and A. Zündorf. Roundtrip engineering with FUJABA. In *Proc. of 2nd Workshop on Software-Reengineering*, 2000.

[23] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2001.

[24] D. Parnas. Information distribution aspects of design methodology. In *IFIP Congress*, 1971.

[25] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, Dec. 1972.

[26] D. Parnas. *Software Fundamentals*. Addison-Wesley, 2001. Edited by Daniel Hoffman and David Weiss.

[27] D. Parnas, P. Clements, and D. Weiss. The modular structure of complex systems. In *Proc. of the 7th Intl. Conf. on Software Engineering*, 1984.

[28] Rational Rose XDE. [http://www-306.ibm.com/ software/ rational].

[29] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Software architecture critics in argo. In *Proc. of the Conf. on Intelligent User Interfaces*, 1998.

[30] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the 24th Intl. Conf. on Software Engineering*, 2002.

[31] V. Sazawal, M. Kim, and D. Notkin. A study of evolution in the presence of source-derived partial design representations. In *In Proc. of the Intl. Workshop on Principles of Software Evolution*, 2004.

[32] N. Scerbakov. Software design project, 2003. [http://coronet.iicm.edu/ sa/ swp_how.htm].

[33] M.-A. D. Storey, H. A. Müller, and K. Wong. Manipulating and documenting software structures. *Software Visualization*, 1996.

[34] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3), 2000.

[35] Tigris.org. Argouml. [http://www.argouml.org].

[36] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[37] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, July 1984.
[38] E. Yourdon and L. Constantine. *Structured Design*. Prentice-Hall, 1979.