

Evaluating Coding Idioms for Object-Oriented Design Patterns using a Pugh Matrix Approach

Srinivas Krishnan
Department of Computer Science
SUNY Brockport
Brockport, NY 14420

Faculty Advisors: Dr Sandeep Mitra and Thomas A. Bullinger¹

¹ArchSynergy, Ltd.
2500 Turk Hill Road
Victor, NY 14564

Abstract

In a paper published in the Proceedings of the 2004 NCUR, the authors proposed a pattern hierarchy approach to object-oriented design that focused on mapping abstract patterns to detailed (design) patterns. This paper extends that research by considering how multiple mappings that exist from abstract to detailed patterns can be evaluated using cohesion and coupling metrics. The hypothesis presented here advocates relative measurement of cohesion/coupling over the use of absolute measures that are hard to obtain in a meaningful way. One mapping between the patterns is categorized as a “base case” and the others are evaluated for cohesion/coupling *relative* to it. A case study to support this hypothesis considers how the Model-View-Controller (MVC) architecture pattern can be implemented by code that uses an implementation framework for the Observer pattern. Two Java frameworks, *java.util.Observer* and Java Beans’ *PropertyChangeListener*, and a third, ‘*Impresario*’ developed by the authors are considered. These are evaluated for cohesion/coupling relative to an identified “trivial” base case. A Pugh matrix approach, as advocated by Design For Six Sigma (DFSS) practitioners, is used in the evaluation process. The results demonstrate the ease with which the proposed approach can be used to select appropriate coding idioms for various patterns.

Keywords: Pattern Hierarchy, Pugh Matrix, Model-View-Controller, Design Patterns, Evaluation

1. Introduction

Every software architect in object-oriented development strives for a design with minimum “reinvention of the wheel.” Design Patterns [4] were supposed to be the next big step in this direction, providing a concrete framework for defining reusability. However the development of this idea has been quite lopsided, ignoring the traceability aspect of applying design patterns to develop code from the analysis and design stages of a software process. The authors in an earlier paper published in NCUR 2004 Proceedings [5] had attempted to bridge this gap and advocated the use of a Pattern Hierarchy approach to object-oriented (OO) software design. This notion of hierarchy was developed to achieve strict *traceability* between the various levels of the model being developed, right up to the code [1,7]. Such traceability is possible when the *problem model* (created at the Analysis phase of the design process) is crafted with precision and documented unambiguously, as prescribed in the Software Engineering Effectiveness Methodology (SEEM) [1]. Uses of patterns, such as a good workflow template, facilitate this goal.

The authors had proposed using patterns in any software process to ensure traceability between the problem (analysis) model, solution (design) and implementation (code) models of the system being developed. However, it was realized that each mapping could result in the discovery of multiple mappings at each level as we go down the

development process. This causes the designer to make a decision at each “node” of the process. This paper hypothesizes that multiple patterns exist to achieve the behavior of a particular mapping, and it is possible to *evaluate* these various patterns to see which one(s) satisfy certain quality goals. Techniques for evaluation are the subject of this paper. As an example, this paper focuses on coding idioms – i.e. the mapping from design models to implementation code. However, the techniques it describes are also applicable to other phases – e.g., in the mapping from analysis to design phase. Techniques for discovering these patterns are beyond the scope of this paper – they can be discovered using *Concept Discovery* techniques mentioned in [1].

2. Metrics

Evaluation of any kind requires one to define a set of metrics that determine the final outcome. Object-oriented development has been concerned with two principal metrics as quality measures of OO design: *Cohesion* and *Coupling* [6]. There have been many investigators in this field who have added to the literature [3,4,6]. However, many of these metrics are borrowed from functional approaches that inadequately measure effectiveness of OO design. Cohesion and coupling are abstract concepts not easy to measure objectively. Any OO system is a set of components (usually, objects) that interact in a defined manner to achieve a certain behavior. The complexity of such interaction is lowered if *high* cohesion components have *low* coupling to each other. Each highly cohesive component is largely self-contained (i.e., works with what it “knows” by itself). And, minimally coupled, it will talk to others less frequently (usually, to get information or a service it should not “naturally know or do”). This paper outlines a technique to measure these features in a typical OO model that includes coding idioms.

It is important to note that the problem domain in this paper is a set of *workflow systems* – i.e., systems with a GUI front-end and a “data store” back-end (e.g., a database, file system) for retrieving and storing persistent data. For the sake of simplicity, the systems are assumed to have a single flow of control (i.e. no concurrency). A workflow system needs a precise depiction of its *behavior* (in addition to its structure). A UML sequence diagram [3] for each *use case* of a system will be the key artifact in this paper.

3. The Pugh Matrix Approach

Absolute measurements of cohesion and coupling of objects/classes are hard to obtain. A discussion of techniques for such measurements is beyond the scope of this paper – however, it must be mentioned that most measurements are based on code eventually written, and not on the process that generated the code. Also, it is hard to relate these measurements to actual consequences. For example, suppose some technique indicates that the coupling level of a particular implemented class is 15. Does this imply that such coupling is bad and the developers should aim for a lower value? Measurements such as these apply only in a *relative* sense. For example, suppose, on some project, two different realizations in code of the same design result in coupling values of 15 and 20. Let these realizations be classes called C1 and C2. The project’s developers then realize that C2 can be taken and used in another context more easily than the former – principally because while it does use (say) five other objects, it only needs to know the interfaces of these objects rather than the implementation of just one other major class C3 that C1 is sensitive to. In the developers’ situation, it is not easy to carry heavyweight classes like C3 around – therefore, even though C1 yielded a lower coupling measure, it is preferable to use C2 in the project environment. Such qualitative judgments are inescapable in finally determining quality.

The approach presented in this paper totally eschews absolute measurements. It focuses on the implementation phase, when designs are realized in implementation code. Consequently, it starts with a set of UML artifacts that illustrate the structure and behavior of the design that is expected to realize a particular use case of the system. For workflow systems, these artifacts must include a UML sequence diagram outlining the workflow associated with the design. The approach then assumes the existence of a “base” realization of this design in code. This realization is one that is traceable to the design, and is usually a *naïve* realization in the sense that it does not use any of the accepted design patterns in its code. Developers then use the Concept Discovery process (by referring to their own past experiences, further research, etc.) to discover other possible realizations in code of the *same* design. These other realizations will use design patterns, and, given that the coding aspect of the development process is being considered, they will also consider widely available *implementation frameworks* corresponding to these design patterns. Once these different realizations are known, they need to be evaluated, *relative to each other*. This is where the Pugh matrix approach used in the manufacturing domain by advocates of the DFSS approach [2] has proved especially useful.

The first step in this process is to identify a set of yardsticks (or quality measures) that are important to the project. Using the Pugh matrix approach, the base realization is considered to have a ‘0’ value for each of these measures. Thereafter, the other realizations are considered for each yardstick, and a judgment about how much better (or worse) it fares for that yardstick vis-à-vis the base realization is made. These judgments are then entered into a Pugh matrix using the codes following the pattern shown in Figure 1.

Judgment	Code
Same as base realization	0
Better than base realization	+
Significantly better than base realization	++
Worse than base realization	-
Significantly worse than base realization	--

Figure 1: Codes for a pugh matrix

A sample Pugh matrix is shown in the abstract in Figure 2.

	Base Realization	Realization 1	Realization 2	Realization 3
Quality Measure 1	0	+	-	0
Quality Measure 2	0	-	++	+
Quality Measure 3	0	--	++	+

Figure 2: Example pugh matrix

Given the judgments shown in Figure 2, the team has to ultimately decide on which realization to use. If Quality Measures 2 and 3 are very important to the project, in the opinion of its stakeholders, then it is quite likely that the team will choose to go with Realization 2, since it obviously fares the best among all possible choices for these measures. Of course, this realization fares worse than the base case in Quality Measure 1. A decision has to be taken about how severe this fact is, and the possible consequences of living with this fact constitute a measure of risk associated with the project that must be captured and documented for effective project management. The rest of this paper focuses on illustrating the use of Pugh matrices for evaluation of coding idioms incorporating design patterns and implementation frameworks with an example.

4. The Example: A “Buy Item” Use Case for a Vending Machine

The example application used to test the evaluation approach was the simulation of a Vending Machine system on a desktop computer. Specifically, the “Buy Item” use case for such a system was considered, where the customer deposits a certain amount of money (in the form of coins) in the machine, and then makes a selection. If the balance in the machine is enough to cover the cost of purchasing one unit of the selected item, then the machine dispenses this item, computes the change to be returned to the customer, returns the change and shows the customer that his/her balance has fallen to zero. The machine is then ready for the next customer.

Figure 3 shows a sequence diagram that depicts the *design-level* workflow associated with this use case. The entities identified in this diagram are the result of using the SEEM process and the pattern hierarchy approach described in [5]. It is not possible to reiterate the details of the process that identified these entities in this paper – however, it should be indicated that the “Vending Clerk” entity is an important abstraction that was identified via the SEEM process, and its existence is central to obtaining the right design. This is mainly because in the design of a workflow system with a GUI front-end, the pattern hierarchy approach indicates that the Model-View-Controller (MVC) pattern must be used. Thus, the Vending Clerk entity serves as the main MVC model object, with which one or more views and controllers are associated. These views and controllers then constitute the GUI with which the human user interacts.

The primary goal of the MVC pattern [4] is to make the classes implementing the pattern more cohesive by distributing workflow responsibilities among the Model, View, and Controller objects. The model is responsible for implementing business logic, the view takes care of displaying results, and the controller handles user input. Many GUI systems combine View and Controller responsibilities in the same object. Figure 3 below shows a workflow among a Vending Clerk model entity and its corresponding View/Controller entity. The main focus in Figure 3 is on

the workflow associated with the “askToBuy” stimulus, which causes the Vending Clerk model to have a responsibility of handling the “askToBuy” request, and as part of handling this request, the View entity gets “called back” several times and is asked to display several different values (such as change, balance, etc.).

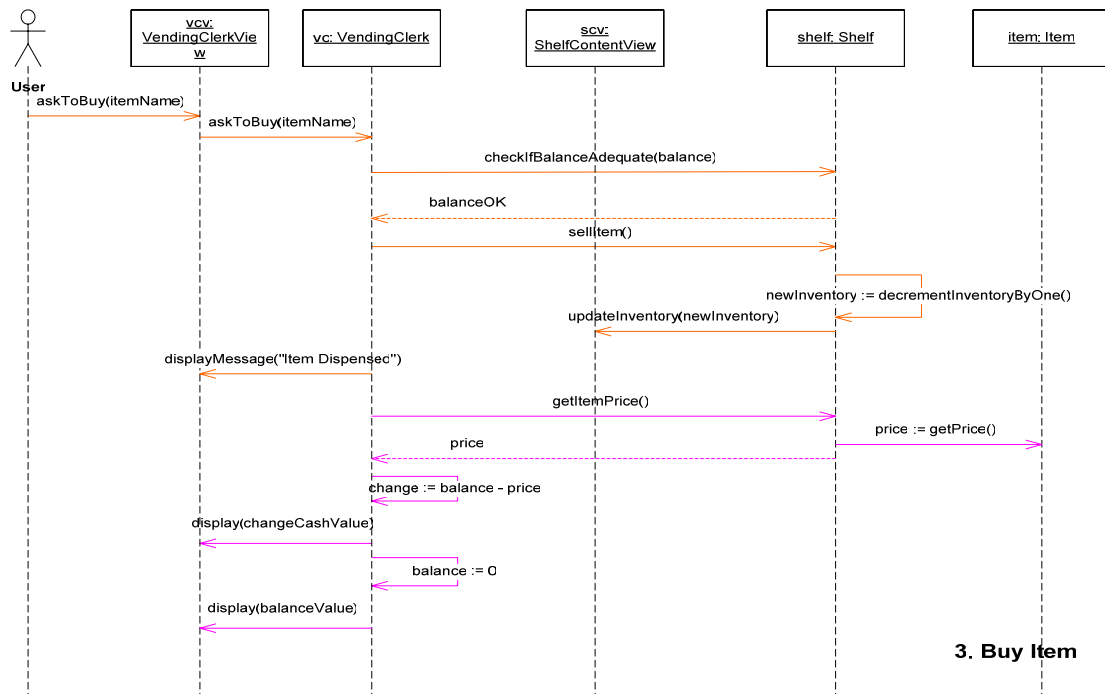


Figure 3: Sequence diagram showing workflow of the “buy item” use case

The “plain vanilla approach” implementing this design has each model and view/controller class having explicit methods to handle these responsibilities. Thus, the view/controller class has methods such as “displayChange (),” “displayBalance (),” etc., each knowing where to display its data in the view. While the use of MVC in the design stage enabled good cohesion in this implementation, it has serious coupling problems. Since the view must know which method in the model to call, the view is coupled to the model class. Moreover, since the model itself must call the various display methods of the view at different times, the model must also know the view’s implementation class, which implies undesirable coupling. The evaluation approach considers this implementation as the “base case”, and seeks alternative implementations that might improve coupling.

Alternatives can be found in the domain of patterns. It is well known that MVC designs may be implemented using *coding frameworks* that enable the implementation of the Observer pattern. Three such frameworks were considered: the framework in standard Java (J2SE) to implement the Observer pattern, Java’s Property Change Listener in the JavaBeans framework, and an author-developed framework entitled “Impresario” [8]. The base case implementation is refactored to include these frameworks and the results of each refactoring compared to the base case.

4.1 using J2SE’s observer design pattern framework

This framework consists of two basic components: the class *java.util.Observable* (message producer) and interface *java.util.Observer* (implemented by each message observer) [6]. Figure 4 shows how these components may be used by the model and view/controllers in the Vending Machine example. The view/controller knows that its associated model (named ‘clerk’) is of class ‘VendingClerk’ and therefore it can invoke the method “askToBuy ()”. The view is still coupled to the model. However, the model itself does not need to know the explicit methods of the View. To the model, the views are simply observers that implement the Java Observer interface and have registered themselves with it. Thus, the model can simply call “setChanged ()” to indicate that it has computed a new value that the observers need to be aware of, and invoke the method “notifyObservers ()” defined in the base Observable class

with the new value. This method *automatically* results in the “update ()” method of the view class being called, in which the update of the display occurs. Thus, the coupling from model to view is reduced.

<pre> class VendingClerk extends Observable { public void askToBuy(String ItemName) { // Compute Change to be handed over setChanged(); //Mark model as having a new value //Notify Subscribed View-Controller of new value notifyObservers(this, "Change value = " + changeValue); } </pre>	<pre> Class VendingClerkView extends JPanel implements Observer { private VendingClerk clerk; ... public void actionPerformed(ActionEvent e) { clerk. askToBuy("candy"); } public void update(Observable obj, Object value) { /* Based on value parameter, change the view to reflect the new values sent by the model */ } } </pre>
--	---

Figure 4: Implementation skeleton using J2SE Observer framework

4.2 using the property change listener infrastructure in the java beans framework

As explained in [6], the Property Change Listener features of the Java Beans framework can be used to realize the Observer design pattern. In this case, the model is an entity that encapsulates a *registry*. This registry contains a mapping from strings known as *keys* to Java-defined *PropertyChangeListener* objects that are interested in that key. For example, the VendingClerkView object could be considered to *subscribe* to the key called “change”, because this object is interested in any new value for this key. This is because it has the responsibility of displaying any new change value. View objects implement the *PropertyChangeListener* interface, and have to ensure that they register themselves properly as shown in Figure 5. The model can simply invoke the “firePropertyChange()” method to indicate that it has computed a new value for a particular key. Figure 5 shows how this works when a new value for change to be returned to the customer is computed. Calling the “firePropertyChange()” method *automatically* results in the “propertyChange()” method for all observers *registered to the key* indicated in the former method being called. It can be noted that this approach also results in the view being tightly coupled to the model, whereas the model is loosely coupled to its view(s). More issues involved using this approach is described in Section 5.

4.3 using the impresario framework

The Impresario framework is described in [8] and also requires the model to have a registry. A major difference in Impresario is that it defines three explicit interfaces – *IModel*, *IView* and *IController*, to be implemented by objects playing these roles. Impresario also establishes an *explicit* notion of state in the model object. Thus, when a model computes the amount of change to be returned to the customer, it updates its own, explicit state component that corresponds to the computed change. Each state component corresponds to a particular *key*, and all view/controllers subscribing to this key get updated. The Impresario framework, just like the Java Beans framework, makes the concept of a model registry explicit. In addition, however, it requires each model to implement an interface, through which controllers can send it (key, value) pairs that change the state of the model, which, in turn, results in subscribing views being updated. Figure 6 shows how this is accomplished. Note that there is no explicit “askToBuy()” method in the model. The functionality is incorporated in the “stateChangeRequest()” method of the *IModel* interface.

<pre> public class VendingClerk { private PropertyChangeSupport registry; public VendingClerk() { //Create Registry for model registry = new PropertyChangeSupport(this); } //Add View to Model Registry public void addPropertyChangeListener(String propertyName, PropertyChangeListener l) { registry.addPropertyChangeListener(propertyName, l); } public void askToBuy(String itemName) { // Compute the amount of change to be returned to customer- // i.e., new value for key named "change" // Notify Views subscribed to "change" Key registry.firePropertyChange("change", null, changeValue); } } </pre>	<pre> public class VendingClerkView implements PropertyChangeListener { private VendingClerk vClerk; public VendingClerkView(VendingClerk clerk) { vClerk = clerk; clerk.addPropertyChangeListener("change", this) } public void actionPerformed(ActionEvent e) { vClerk..askToBuy("candy"); } public void propertyChange(PropertyChangeEvent evt) { String propertyChangeKey = evt.getPropertyName(); if (propertyChangeKey.equals("change") == true) { String value = (String)evt.getNewValue(); // show this value (of change) in appropriate place in view } } } </pre>
---	--

Figure 5: Implementation skeleton using Java Beans framework

5. Evaluation of the Idioms using a Pugh Matrix

Each of the coding idioms resulting from the above three frameworks are evaluated on the basis of the following quality measures: *Cohesion*, *Replacability*, *Scalability*, *Extensibility*, and *Setup*. *Replacability* refers to the ease with which components in the implementation can be changed. For example, a new view class that has better aesthetics may be developed and it should be possible to make the same model seamlessly interact with these new view classes. *Scalability* refers to the ease with which multiple views of the same kind, perhaps on different display hardware, can be added to the system as listeners to the same model. *Extensibility* refers to the ease with which new business logic can be incorporated in both the model and view components. For example, suppose that the model also computes total sales for the day, and some (but maybe not all) views need to show the total sales. *Setup* refers to the total code that needs to be written to set up the model and its subscribing view/controllers before the system can be put into operation. Note that Replacability, Scalability and Extensibility are coupling-oriented measures.

Judgments made about these quality measures after extensive considerations of the three idioms are reflected in the Pugh matrix of Figure 7. Note that the matrix reflects judgments relative to the base case, which has all '0' entries. Replacability is easier in J2SE Observer and Java Beans because the model is not tightly coupled to its views as in the base case. In both J2SE Observer and Java Beans, use of Java frameworks for the Observer design pattern eliminated the need for the model to know that it has to call explicit methods in the view like "displayChange()", "displayBalance()", etc. to show different values in different locations of the view. Rather, the model called a Java-defined method that automatically arranged for a standard method in the view to be called, and this latter method extracted from its parameters the new values and the new locations to display these values. However, both these approaches still couple the view tightly to its model. Impresario, through the notion of the *IModel* interface, allows looser coupling from view to model, and hence scores higher.

The concept of *keys* to represent a particular kind of value, which is present in the Java Beans and Impresario frameworks, enhances Scalability. Views are called back several times – e.g., to display change, display new balance, display which item just sold, etc. It is possible that in cases with multiple views, some (possibly remote) views will only be interested in showing the current inventory of items, and not change rendered, etc. In the case of using J2SE Observer, *all* observers will be automatically called back for *every* new value computed by the model. On the other hand, in Java Beans and Impresario, *only* those observers subscribing to that key will be called back.

<pre> public class VendingClerk implements IModel { //Model Maintains registry of interested objects private Registry myRegistry; public VendingClerk() { myRegistry = new ModelRegistry(); // Set dependencies to show that buyItem interaction // results in new change and balance computation myRegistry.setDependency("buyItem", "change, balance"); } public void subscribe(String key, IView subscriber) { myRegistry.subscribe(key, subscriber); } // Method to return state values computed // e.g., change to return, balance etc. public String getState(String key) { if (key.equals("change") == true) { // return new computed change value } ... } public void stateChangeRequest(String key, Object value) { /* Based on key check the value and update state */ if (key.equals("buyItem") == true) { // Compute new value for change, balance, etc. // Use registry to update all those who are interested // in new change value and balance value (because, as // the registry dependency shows, these depend on the // "buyItem" interaction coming in to this model . myRegistry.updateSubscribers(key, this); } } </pre>	<pre> public class VendingClerkView implements IView { public VendingClerkView() { //Subscribe to model, saying interested in key change myModel.subscribe("change", this); } public void actionPerformed(ActionEvent evt) { if (evt.getSource() == buyButton) { myModel.stateChangeRequest("buyItem", "candy"); } } public void updateState(String key, Object value) { /*Based on key check the value and change the view to reflect change in Model state*/ if (key.equals("change") == true) { // Use value parameter to display new change value } } } </pre>
---	--

Figure 6: Implementation skeleton using Impresario framework

	Base Case	J2SE Observer	Java Beans	Impresario
Cohesion	0	0	0	0
Replacability	0	+	+	++
Scalability	0	0	+	+
Extensibility	0	+	++	++
Setup	0	0	-	-

Figure 7: Final Pugh matrix

Thus, when the model computes which item is sold, only those views subscribing to the “soldItem” key will be called, whereas when it computes the amount of change to be rendered, only those views subscribing to the “change” key will be called. This makes for better scalability/performance – hence, the higher ratings in the Pugh matrix.

Similar arguments apply for the Extensibility measure. Space considerations do not enable the presentation of details as to why extensibility scores are higher in the case of Java Beans and Impresario – they relate to the way in which the code of the model has to be changed to accommodate the new functionality. Briefly, it may be mentioned that the J2SE Observer requires the “setChanged ()” method to be called before *every* new call to “notifyObservers ()”, and this results in J2SE Observer scoring lower. Finally, because of the need for views to *explicitly subscribe* to various keys in the model, setup code is more complex in Java Beans and Impresario, resulting in them scoring lower for these measures.

The values in the Pugh matrix above indicate that in a project where there are many different types of views, all subscribing to the same model, but the model itself may change (say, due to new functionality being introduced), the Impresario framework is the best one to use. Views can then be replaced incrementally as needed. On the other hand, if there are many views using the same model, and each view uses different subsets of the data from the model, then using the Java Beans and Impresario frameworks are better performance-wise, because only those views interested in a particular piece of data will get called. Such incremental calls are impossible in the J2SE Observer case, and hard to achieve in the base case, which is not set up at all to handle multiple views, and would require considerable rework. Similar judgments can be made about the other quality measures.

6. Conclusions and Future Work

The Pugh matrix approach has proved to be useful in evaluating alternative coding idioms on the basis of a set of quality measures important to a project. Future work could involve using this approach to evaluate alternative coding idioms for other design patterns such as Factory, Mediator, etc. Also, the authors are interested in exploring how this approach could be applied to earlier phases of the life cycle, such as evaluating alternative designs corresponding to a particular analysis model.

7. References

1. Bullinger, Thomas, Mitra, Sandeep (March 2004). A holistic approach to Embedded Systems Development. Proceedings of the Embedded Systems Conference (West), San Francisco, CA (available on CD-ROM).
2. Creveling, Clyde, Slutsky, Jeffrey, Antis D. (2002). Design for Six Sigma in Technology and Product Development. Upper Saddle River, NJ: Prentice Hall.
3. Fowler, Martin (2004) UML Distilled: A Brief Guide to the Standard Object Modeling Language. Boston, MA: Addison-Wesley – Pearson Education.
4. Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.
5. Krishnan, Srinivas, Mitra, Sandeep, Bullinger, Thomas (2004). A Pattern Oriented Approach to Software Systems Design. Proceedings of the National Conference on Undergraduate Research (available on CD-ROM).
6. Larman, C. (2000) Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Upper Saddle River, NJ: Prentice-Hall.
7. Mitra, Sandeep, Rao T M, Bullinger, Thomas (April 2005) Teaching Software Engineering using a Traceability-based Development Methodology. Proceedings of the CCSCNE 2005 Conference, Providence, RI (to appear).
8. Mitra, Sandeep, Bullinger, Thomas (February 2004) The Impresario Framework: Effective Realization of Model-View-Controller Implementations. ArchSynergy Technical Report.