Design Pattern Directed Clustering for Understanding Open Source Code

Zhixiong Han, Linzhang Wang, Liqian Yu, Xin Chen, Jianhua Zhao and Xuandong Li
State Key Laboratory of Novel Software Technology
Department of Computer Science and Technology, Nanjing University
Nanjing, Jiangsu, P.R.China 210093
{hzx,yuliqian}@seg.nju.edu.cn, {lzwang,chenxin,zhaojh,lxd}@nju.edu.cn

Abstract

Program understanding plays an important role in the reuse and maintenance of open source code. Rapid evolving and bad documentation makes the understanding and reusing difficult. Many software structure clustering techniques have been proposed to produce and recover architectural views directly from source code to facilitate the understanding activities. However, most of them focus on structural information and neglect deep design information such as design pattern relationships which might contribute more in the understanding. Design patterns are widely employed in software development, especially in open source projects. In this paper, we propose a design pattern directed clustering approach to help understand open source code. First, we use a well known design pattern detection tool to detect design pattern instances in the source code under consideration. Then, we treat classes as players of design pattern instances, and group them into clusters according to the inherent association among their participating design pattern roles. Last, we visualize the generated clusters. We have implemented a prototype tool TasteJ. To justify our approach, we also conducted an experiment on an open source system JHotDraw6.0 with TasteJ and a state of art structure clustering tool Bunch respectively. The preliminary experimental results show that our approach is feasible and promising.

1. Introduction

Program comprehension plays an important role in software engineering activities of design, development and maintenance. Many famous cognitive models have been proposed to explain how programmers comprehend software [23]: top-down comprehension model, bottom-up comprehension model, opportunistic and systematic strategies comprehension model, etc.

As the modern software growing large and complex, open source becomes more and more popular nowadays. People contribute and reuse open source code in academia, industry and open source community. Open source code evolves quite rapidly, but it often lacks of documentation or the documents are out-of-date, which makes it hard to comprehend, reuse and maintain.

In order to provide high-level architectural views to facilitate the understanding activities, based on the top-down comprehension model and the “divide and conquer” principle, many software structure clustering approaches have been proposed [15, 18, 16, 20, 13, 11]. They aim at identifying areas of a system that are loosely coupled to each other. Most of them treat the software structure clustering (partition) as a search or optimization problem where hill-climbing, simulated annealing, genetic algorithms, etc., are used [18, 16, 20, 13]. Some approaches consider program source code as plain text, and group software entities based on similar terms [11, 14], e.g., file names, code identifiers and comments. The file ownership [1] and change history [2] are also utilized by some researchers. All these approaches have shown their merits, yet most of them focus on structural information, and ignore deep design information such as design pattern relationships which might contribute more in the understanding.

It is well known that design patterns [7] are widely used in practice, especially in the open source community. This paper proposes a new clustering approach according to design pattern relationships which are deemed to contain both structural and design information. First, we use existing design pattern detection techniques and detectors, e.g., PINOT [21] to help recover the design pattern relationships from concerned programs. Then, we treat clustered classes as *player*s (i.e., *participating class*es) of design pattern instances [29], and group the classes into different clusters according to the inherent association among their participating design pattern *role*s. Last, we visualize the generated clusters.

The contributions of this paper are listed below.

- We use design pattern information to direct the software structure clustering process;

- We develop a tool to automate the clustering process and visualize the clustering results.

The rest of this paper is organized as follows. Section 2 introduces the background including design pattern detection techniques and presents a motivating example. Section 3 illustrates our proposed design pattern directed clustering approach. Section 4 presents the experiment on JHotDraw6.0. Section 5 reviews the related work, and Section 6 concludes the paper.

2. Background and Motivation

2.1. Design pattern detection techniques

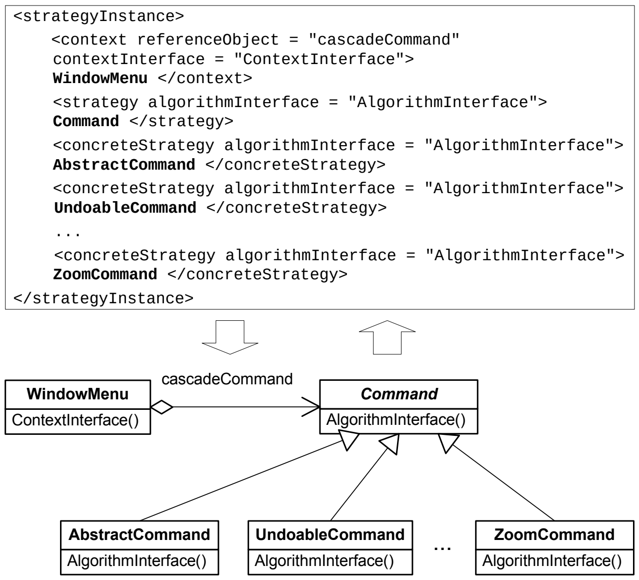
A design pattern abstracts a reusable object-oriented design that solves a common recurring design problem in a particular context [7]. According to literature [21], the 23 GoF patterns can be classified in a reverse-engineering sense, into five categories: language-provided patterns, structure-driven patterns, behavior-driven patterns, domain-specific patterns and generic concepts. The remaining of this paper only takes two categories of them into consideration:

- Structure-driven patterns that are driven by code structure, including Bridge, Composite, Adapter, Facade, Proxy, Template Method and Visitor; and
- Behavior-driven patterns that are driven by system behavior, including Singleton, Abstract Factory, Factory Method, Flyweight, Chain of Responsibility, Decorator, Strategy, State, Observer and Mediator.

In order to reveal the intent and design of a software system, many design pattern detection approaches and tools have been designed to directly extract design pattern instances from source code (or Java byte code). These approaches identify structure-driven patterns mainly by analyzing inter-class relationships, and detect behavior-driven patterns by using machine learning, dynamic analysis and static program analysis [21]. Some popular design pattern detection tools for specific languages are listed below.

- For C++: SPQR [22], DPRE [3], Columbus [6], the detector described in [28], etc;
- For Java: Ptidej [10], FUJABA [19], PINOT [21], DP-Miner [4], Scoring [26], etc;
- For SmallTalk: SOUL [5], etc.

A detected design pattern instance (DPI) refers to the participating components (e.g., class, method, member variable) that work together to satisfy the requirements of a certain design pattern. Figure 1 describes the detail of a DPI using both XML format and class diagram format. The DPI is extracted from the detection result produced by the tool PINOT, with JHotDraw6.0 beta1 [9] as input. It satisfies the requirements of the Strategy design pattern. For brevity, the DPI can be noted as DPI(context: `WindowMenu`; strategy: `Command`; concreteStrategy: {`AbstractCommand`, `UndoableCommand`, `ZoomCommand`}), where class `WindowMenu` is the player of the role context, `Command` the player of the role strategy, `AbstractComand`, `UndoableCommand` and `ZoomCommand` the players of the role concreteStrategy. Other components are neglected as we concern the information on participating classes only.



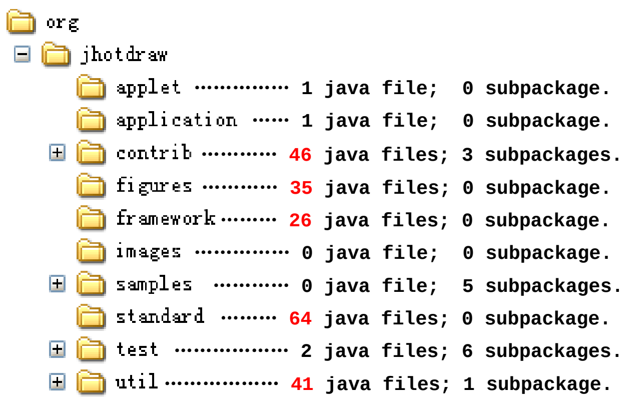
```
<strategyInstance>
    <context referenceObject = "cascadeCommand"
    contextInterface = "ContextInterface">
    WindowMenu </context>
    <strategy algorithmInterface = "AlgorithmInterface">
    Command </strategy>
    <concreteStrategy algorithmInterface = "AlgorithmInterface">
    AbstractCommand </concreteStrategy>
    <concreteStrategy algorithmInterface = "AlgorithmInterface">
    UndoableCommand </concreteStrategy>
    ...
    <concreteStrategy algorithmInterface = "AlgorithmInterface">
    ZoomCommand </concreteStrategy>
</strategyInstance>
```

Figure 1. A Strategy pattern DPI

2.2. A motivating example

In this paper, open source medium-size project JHotDraw6.0 beta 1 is taken as the subject program which we would try to uncover along with the remaining discussion of this paper. JHotDraw is a two-dimensional GUI framework for structured drawing editors [9]. This released version is written in Java, consisting of 484 Java files, about 70KLOC. Its high-level package structure is outlined in Figure 2.

The high-level code storage structure of JHotDraw6.0 is well designed. `Org` on the first layer and `jhotdraw` on the second layer point out the property of the development organization and the project name respectively. On the third layer the entire system is divided into 10 packages, whose names take on their respective functionalities. For example,



```
org
jhotdraw
  applet ·············· 1 java file;  0 subpackage.
  application ······ 1 java file;  0 subpackage.
  contrib ··········· 46 java files; 3 subpackages.
  figures ··········· 35 java files; 0 subpackage.
  framework ········ 26 java files; 0 subpackage.
  images ·············· 0 java file;  0 subpackage.
  samples  ··········· 0 java file;  5 subpackages.
  standard  ········ 64 java files; 0 subpackage.
  test  ················· 2 java files; 6 subpackages.
  util ················· 41 java files; 1 subpackage.
```

Figure 2. High-level package structure of JHotDraw6.0 beta 1

package `applet` and package `application` provide standard UI for JHotDraw applets and applications respectively, package `image` stores image resources, package `samples` and package `test` answer for samples and testing respectively. Obviously, these packages just provide assistant and peripheral entry functions for the system. They are not the point where core functionality exists.

In fact, the core function of this software system lies in the rest five big packages, i.e., `contrib`, `figures`, `framework`, `standard` and `util`. They have a much bigger size than other packages, consisting of 46, 35, 26, 64 and 41 classes, respectively.

Unfortunately, no matter which package to start from, we would immediately encounter one common problem, i.e., the number of classes contained by a package is so enormous that we even had no idea which class or classes should be inspected first. This is not an accidental case, which happens to many modern software systems. Based on the "divide and conquer" principle, we plan to partition every big package into smaller clusters, so as to reduce each inspection's scope, and alleviate the intellectual effort.

We use our designed clustering approach to cluster the classes of every big-size package above respectively. The produced clustering results provide reasonable and helpful layouts for us to understand the internal structure of the packages. Related clustering results would be presented in the experiment section (Section 4).

3. Design pattern directed approach

We suggest a design pattern directed comprehension assistance approach to aid software developers and maintainers in understanding the open source code implemented under the guidance of design pattern intents, by clustering classes based on the inter-role relationships.

3.1. Approach

The overview of our approach is shown in Figure 3. It consists of three main components: the design pattern detector, the design-pattern-role-based (DPRole-based) clustering module and the visualization module. The design pattern detector accepts open source code and generates DPIs. The DPRole-based clustering module, which is the core component of the approach, groups classes into clusters using the DPRole-based clustering algorithm (Section 3.2). The algorithm needs two inputs, one is the classes that are concerned by the user, e.g., the classes in a package or directory, the other is the classes' related DPIs which are recorded in XML format. With these inputs in place, the algorithm outputs partitioned clusters. The visualization module visualizes the partitioned clusters so that they can be easily and conveniently inspected.

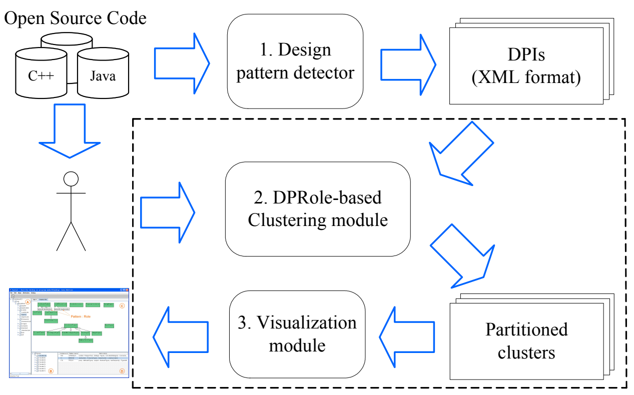


Figure 3. Overview of design pattern directed clustering approach

3.2. DPRole-based clustering algorithm

We treat concerned classes as players of DPIs and group them into different clusters according to the inherent association among their participating roles inspired from the detected DPIs. We find that though players of a DPI always work together for achieving a common goal, i.e., the target pattern's intent, the degree of their internal functional correlations can be very different.

As the Strategy pattern DPIs in Figure 4, the players of the role strategy and concreteStrategy always maintain a high degree of correlation, e.g., for DPI1 (DPI2), the players describe "tools", and for DPI3, the players describe "commands". Whereas, the degree of correlation between the players of the role context and strategy (concreteStrategy) seems much weaker, e.g., for DPI1 and DPI2, though they have common players on strategy (concreteStrategy),

DPI1(context: EventDispatcher; strategy: Tool; concreteStrategy: {CreationTool, SelectionTool, ConnectionTool, ...});

DPI2(context: JavaDrawViewer; strategy: Tool; concreteStrategy: {CreationTool, SelectionTool, ConnectionTool, ...});

DPI3(context: EventDispatcher; strategy: Command; concreteStrategy: {CopyCommand, CutCommand, PasteCommand, ...}).

Figure 4. Three similar Strategy pattern DPIs

their players of context can be freely exchanged, i.e., `EventDispatcher` and `JavaDrawViewer`. Likewise, for DPI1 and DPI3, though they have common players on context, their players of strategy (concreteStrategy) can be greatly distinct. As to what make these happen, we concluded that because there inherently exists different degrees of correlations between various design pattern roles, which might be useful for guiding the software structure clustering process.

Pattern rules and close-role sets. Aspired by above discovery, we cluster classes based on *close-role set*s, which are declared by *pattern rule*s so as to indicate the intrinsic high degree correlations among design pattern roles. Suppose $DPI(r1 : c1; r2 : c2; r3 : c3; r4 : c4; \dots)$ is an instance of the pattern dp, whose pattern rule has declared a close-role set $\{r1, r2, r4\}$, and then a cluster $\{c1, c2, c4\}$ can be produced according to the set. It is worth noting that not every role of a DPI must be putted into a certain close-role set, e.g., $r3$. Before explaining our clustering algorithm in detail, next, we glad to define pattern rules and explain close-role sets for structure-driven patterns and behavior-driven patterns.

Structure-driven patterns establish the architecture among classes but do not specify the behavioral details for them. The Bridge pattern of this category is designed to decouple an abstraction from its implementation so that the two can vary independently on the responsibility [7]. However, the functional correlation between its role abstraction (refinedAbstraction) and the role implementor (concreteImplementor) remains intimate and compact. They depict a common stuff at different abstract levels, and abstraction always maintains a reference of the implementor to make a connection between them. We grouped these closely correlated roles into a close-role set through defining a related pattern rule as follows. The rule is also summarized in the first row of Table 1 (Other pattern rules shown in Table 1 have similar definitions).

Bridge Rule. *Assume there is a Bridge design pattern instance, DPI(abstraction: $ClassA$; $refindAbstraction$: $SetA$; implementor: $ClassI$; $concreteImplementor$: $SetI$), in which $ClassA$ is a player of the role abstraction, $SetA$ a collection composed by players of the role $refindAbstraction$, $ClassI$ a player of the role implementor, and $SetI$ a collection composed by players of the role $concreteImplementor$. Then there is a close-role set $\{abstraction, refindAbstraction, implementor, concreteImplementor\}$, whose corresponding close-player set is $\{ClassA, SetA, InterfaceI, SetI\}$ (that is a set constituted by $ClassA$, $InterfaceI$, elements in $SetA$ and elements in $SetI$).*

For the Composite pattern, it is designed to compose objects into tree structures to represent part-whole hierarchies [7]. The role component declares a public interface for both individual objects and compositions, and the role composite represents a composite component. Generally, they have similar function definitions and frequent interactions that need to be taken into consideration. Therefore, we put them into a close-role set, as shown in the second row of Table 1. However, we did not plan to bring in the role leaf, as it usually owns many other private features except complying with the interface declared by the component. The pattern rules and close-role sets for the Adapter, Proxy, Template Method, and Visitor patterns are also given in Table 1. It is allowed to revise them or add new ones as needed in terms of personal understanding. It is worth mentioning that all above close-role sets do not comprise the role client since that it usually merely represents an entry for specific patterns and a detector hardly takes it into account during the DPI recovery process.

Different from the structure-driven patterns, behavior-driven patterns require specific actions implemented in the method bodies. Usually, most of these patterns are used just because of some specific aspects of the participants' behavioral requirements, not the reflection of their main functional requirements. For instance, the Observer pattern is used just because of the "observing" needs from observers to subjects, the Chain of Responsibility pattern used merely because of the "request delivery" needs of some chained objects, the Mediator pattern used just because "colleagues" need a "mediator" to coordinate their communication and the Decorator pattern used because a public "wrapper" is needed.

For the Strategy pattern, we have learned that strategy and concreteStrategy always maintain a high degree of correlation, so they ought to be grouped into a close-role set. The structure of the State pattern is close to the Strategy pattern. The difference between them is that the former is used to depict "states", and the latter is used to describe "strategies" (i.e., algorithms). Therefore, the close-role set declared for the State pattern is similar to that of the Strategy pattern, as we can see in the last two rows of Table 1.

Our clustering algorithm incrementally groups the classes of a package or directory into clusters in a bottom-

Table 1. Pattern rules and close-role sets

Pattern Rule	DPI	Close-role set(s)	Close-player set(s)
Bridge[1]	DPI(abstraction: ClassA; refindAbstraction: SetA; implementor: ClassI; concreteImplementor: SetI)	{abstraction, refindAbstraction, implementor, concreteImplementor}	{ClassA, SetA, InterfaceI, SetI}
Composite[1]	DPI(component: ClassC1; composite: ClassC2; leaf: SetL; client: ClassC3)	{component, composite}	{ClassC1, ClassC2}
Adapter[1]	DPI(adapter: ClassA1; adaptee: ClassA2; target: ClassT; client: ClassC)	{adapter, adaptee, target}	{ClassA1, ClassA2, ClassT}
Proxy[1]	DPI(proxy: ClassP; subject: ClassS; realSubject: SetS)	{proxy, subject, realSubject}	{ClassP, ClassS, SetS}
Template Method[1]	DPI(abstractClass: ClassC1; concreteClass: ClassC2)	{abstractClass, ConcreteClass}	{ClassC1, ClassC2}
Visitor[1]	DPI(visitor: ClassV; element: ClassE; concreteVisitor: SetV; concreteElement: SetE; objectStructrue: ClassS)	{visitor, concreteVisitor} {element, concreteElement}	{ClassV, SetV} {ClassE, SetE}
Strategy[2]	DPI(context: ClassC; strategy: ClassS; concreteStrategy: SetS)	{strategy, concreteStrategy}	{ClassS, SetS}
State[2]	DPI(context: ClassC; state: ClassS; concreteState: SetS)	{state, concreteState}	{ClassS, SetS}

[1] Rules on structure-driven patterns [2] Rules on behavior-driven patterns

up manner based on the assumption that if class A have a close role correlation with class B, and class B also have a close role correlation with class C, then putting A, B and C together would be beneficial for understanding all of them. The algorithm is described in Figure 5.

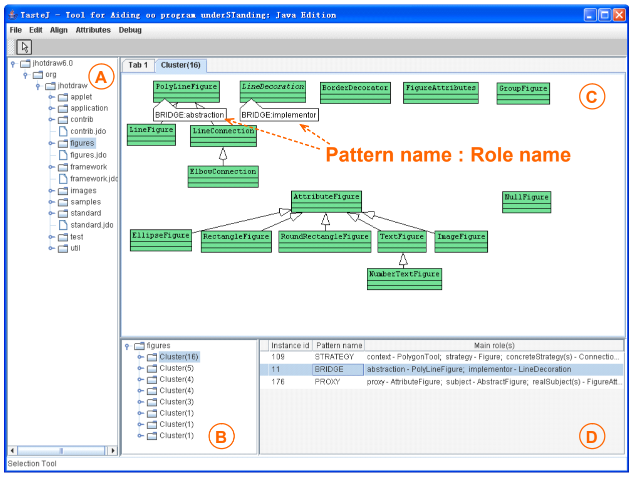
```
Input: $S_{class}, S_{DPI}$.
Output: A set of partitioned clusters $S_{par\_cluster}$.

DPRoleBasedClustering($S_{class}$, $S_{DPI}$)
  $S_{cluster} \leftarrow S_{class}$   // Initially
  // Begin the clustering process
  for each $DPI \in S_{DPI}$ do
    if $DPI.type \in T$ then
      // Get close-role sets for the $DPI.type$ pattern.
      $S_r \leftarrow getCloseRoleSets(DPI.type)$
      for each $S_{close\_role} \in S_r$ do
        // Count corresponding close-player set.
        $S_{close\_player} \leftarrow count(S_{close\_role}, DPI)$
        // Omit redundant players from $S_{close\_player}$.
        $S_{key\_player} \leftarrow omit(S_{close\_player}, S_{class})$
        // Merge clusters that contain players in $S_{key\_player}$.
        $S'_{cluster} \leftarrow merge(S_{cluster}, S_{key\_player})$
        $S_{cluster} \leftarrow S'_{cluster}$
      end for
    end if
  end for
  // Here, $S_{cluster}$ holds the partitioned result.
  $S_{par\_cluster} \leftarrow S_{cluster}$
  return $S_{par\_cluster}$
```

Figure 5. DPRole-based clustering algorithm

The algorithm inputs a class set S_{class}, and a DPI set S_{DPI} consisting of the detected DPIs that relate to the classes in S_{class}. Initially, for each class in S_{class}, construct a cluster, let the cluster contain the class only (such a cluster which contains only one member is called *unit cluster*), and all the initial unit clusters make up a cluster set $S_{cluster}$. In the clustering process, for each DPI in S_{DPI}, if its type is one of the concerned design pattern types, i.e., $DPI.type \in T$, $T = \{bridge, composite, adapter, proxy, visitor, template\_method, strategy, state\}$, get the design pattern's related close-role sets from Table 1. Then, for each close-role set S_{close_role}, merge the clusters that contain players for the roles in it. Concretely, first, count its corresponding close-player set S_{close_player}. Second, omit the players which are not concerned, i.e., not covered by S_{class}, from S_{close_player}, the remaining close players are collected in S_{key_player}. Third, merge the clusters that contain players in S_{key_player}. Last, update cluster set $S_{cluster}$ with the newly generated cluster set $S'_{cluster}$. After processing every input DPIs, $S_{cluster}$ holds the partitioned clusters, which are assigned to $S_{par\_cluster}$ as output.

3.3. Implementation

We have developed an automated comprehension assistance tool TasteJ [25] (Tool for Aiding program underSTanding: Java Edition) according to the clustering approach and techniques proposed in this paper. It is implemented using Java based on the well-designed drawing framework JHotDraw, so it can be easily expanded and migrated. It has realized the contents rounded by the dashed line in the approach overview figure (Figure 3). TasteJ accepts concerned Java classes and their related DPIs, and displays generated result in graphical mode. The user can interactively browse the result. The GUI of it is shown in Figure 6. Region A displays the file structure of a currently considered system, region B displays the generated clusters with respect to the package selected in region A. Region C displays the internal structure of a selected cluster using a simple class diagram that paints inheritance relations only; at the same moment, region D lists the cluster's related *relied DPIs* (a DPI is considered as a relied DPI if and only if it is exactly useful for directing classes grouped into a common cluster). While one of the listed relied DPIs is selected, the currently considered package's members who act roles in it would be labeled with corresponding pattern names and role names, as indicated in Figure 6.



Instance id	Pattern name	Main role(s)
109	STRATEGY	context - PolygonTool; strategy - Figure; concreteStrategy(s) - Connectio...
11	BRIDGE	abstraction - PolyLineFigure; implementor - LineDecoration
176	PROXY	proxy - AttributeFigure; subject - AbstractFigure; realSubject(s) - FigureAtt...

Figure 6. GUI of TasteJ

In the development of TasteJ, We applied preprocessing to the inputted DPIs in order to improve the execution efficiency of the clustering algorithm, and postprocessing to the outputted clusters so as to refine the result.

Preprocessing. The output of a design pattern detector is often a series of raw DPIs. We can merge the DPIs that have both the same pattern type and players for the roles covered by a close-role set to reduce the number of DPIs the algorithm need to process. For instance, the DPI1 and DPI2 described in Figure 4 can be merged into a new instance, DPI(context: {`EventDispatcher`, `JavaDrawViewer`}; strategy: `Tool`; concreteStrategy: {`CreationTool`, `SelectionTool`, `ConnectionTool`, ...}).

Postprocessing. Sometimes, the algorithmic output may contain too many unit clusters , e.g., the clustering result of package `standard` that would be discussed in the experiment section(Section 4), which contains 28 clusters in total, 21 of which are unit clusters. In this case, the input package is excessively divided. Obviously, it is not easy to handle too many fragments. To deal with this problem, we are able to use inheritance and sibling relationship to further process the output as follows.

1. For a unit cluster, if a non-unit cluster exists and the only member of the unit cluster has an inheritance or sibling relationship with one of the non-unit cluster's members, we take the only member of the unit cluster to the non-unit cluster to form a new cluster and delete the original unit cluster from the global cluster set.

2. After the above process, if the number of the unit clusters is still more than default size 10, we merge the unit clusters whose members hold an inheritance or sibling relationship with each other.

4. Experiment

We conducted an experiment on the project JHotDraw6.0, and compared our experimental results with the results produced by the well-known software structure clustering tool Bunch [15, 18].

4.1. Clustering results

This subsection describes the clustering results generated by our tool TasteJ using those big packages contained by JHotDraw6.0 and their related DPIs which are detected by the tool PINOT as input. The clustering results of package `figures` and `standard` are explained in detail, and the results of package `contrib`, `util` and `framework` are briefly summarized.

Clustering result of package figures. The partitioned clusters of package `figures` are shown in Figure 7. Its total 35 classes (and interfaces) is divided into 8 clusters, in which, Cluster 1 contains 16 classes, Cluster 2, 3, 4, 5, 6, 7 and 8 contain 5, 4, 4, 3, 1, 1 and 1 classes respectively. The size of Cluster 1 seems a bit large, whereas its contained classes apparently have a common theme, i.e., "figure". Classes `EllipseFigure`, `LineFigure`, `PolyLineFigure`, `RectangleFigure` and `RoundRectangleFigure` depict basic geometric "figures". `ElbowConnection` and `LineConnection` depict connection "figures". `BorderDecorator` and `LineDecoration` depict decoration "figures". `AttributeFigure`, `ImageFigure`, `NumberTextFigure` and `TextFigure` depict purpose-oriented "figures", each of which can be used in a particular context. The remaining classes of cluster 1 render related services for these "figures", `GroupFigure` supplies a way for grouping them, `NullFigure` provides a default implementation for them, and `FigureAttribute` offers a container for the attributes of them. Similar to Cluster 1,

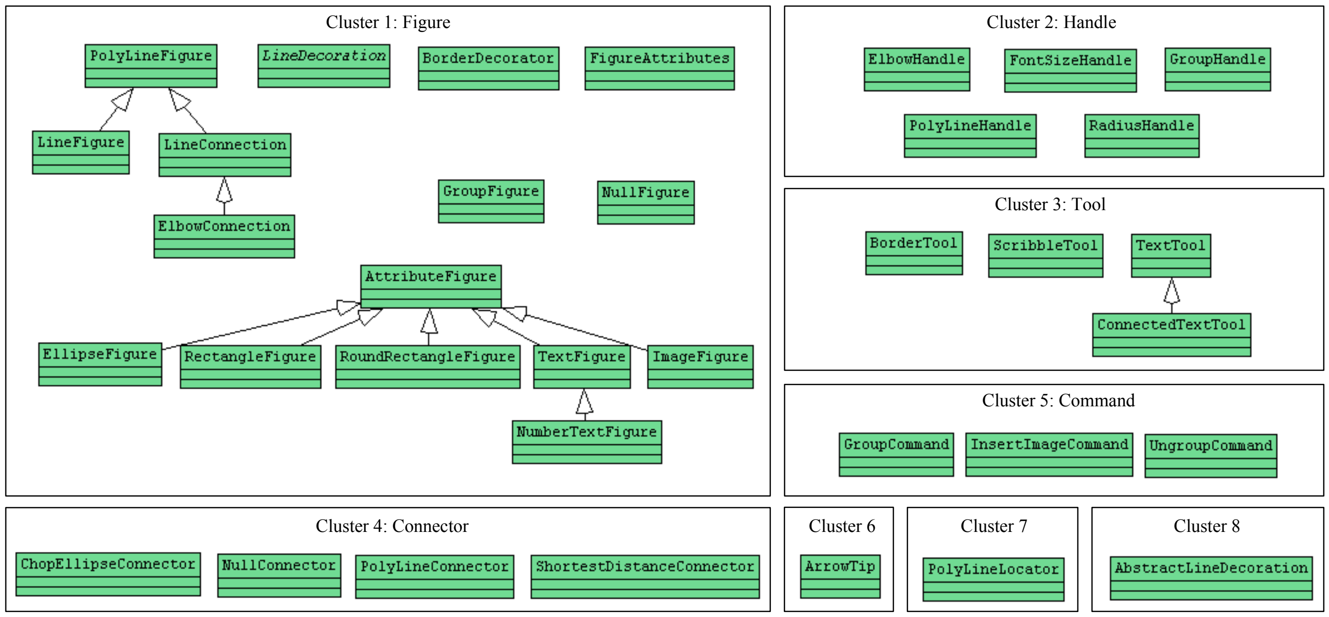


Figure 7. Partitioned clusters of package figures (generated by TasteJ)

Cluster 2, 3, 4 and 5 also have their respective described themes, i.e., “handle”, “tool”, “connector” and “command”. The other three clusters are unit clusters. They will be explained latter. Next, the clustering result of a larger package is presented.

Clustering result of package standard. Because of the space issue, we list package `standard`’s partitioned clusters (that generated before postprocessing) below instead of drawing them.

- Cluster 1 : AbstractCommand, AlignCommand, ChangeAttributeCommand, BringToFrontCommand, CopyCommand, CutCommand, DeleteCommand, DuplicateCommand, FigureTransferCommand, PasteCommand, SelectAllCommand, SendToBackCommand, ToggleGridCommand;
- Cluster 2 : AbstractTool, ActionTool, ConnectionTool, CreationTool, DragTracker, HandleTracker, NullTool, SelectAreaTracker, SelectionTool;
- Cluster 3 : AbstractHandle, ChangeConnectionEndHandle, ChangeConnectionHandle, ChangeConnectionStartHandle, ConnectionHandle, LocatorHandle, NullHandle;
- Cluster 4 : AbstractFigure, CompositeFigure, DecoratorFigure, StandardDrawing, QuadTree;
- Cluster 5 : AbstractLocator, OffsetLocator, PeripheralLocator, RelativeLocator;
- Cluster 6 : AbstractConnector, ChopBoxConnector, LocatorConnector;
- Cluster 7 : NullDrawingView, StandardDrawingView;
- Cluster 8 to 28: each contains only one class.

Package `standard` that contains 64 classes is divided into 28 clusters, in which, Cluster 1, 2, 3, 4, 5, 6 and 7 contains 13, 9, 7, 5, 4, 3 and 2 classes respectively, the other are unit clusters. The former seven clusters concentrate on “command”, “tool”, “handle”, “figure”, “locator”, “connector”, and “drawing view”, respectively (noting that: in Cluster 2, `DragTrack` is a tool that implements the dragging of a clicked figure, `HandleTracker` is a tool for interactions with the handles of a figure, and `SelectAreaTrack` is a tool for the rubber hand selection of an area; in Cluster 4, `StandardDrawing` is a composite figure and `QuadTree` is an assistant data structure for drawing figures). These clusters already take up 43 of all 64 classes. Unfortunately, the remaining classes are not properly grouped, making the number of unit clusters grow rapidly. We reduced the number by further processing the partitioned clusters in accordance with the inheritance and sibling relationship among their contained classes, as discussed in Section 3.3.

After postprocessing, the number of unit clusters greatly decreases from 21 to 7. Meanwhile, the number of total clusters reduces from 28 to 18, whereas the impact on the revealed described themes is tiny. As postprocessing mainly repartitions the unit clusters’ members into existing non-unit clusters, the non-unit clusters’ main components would not change much.

In fact, we are also able to refine the clustering result of package `figures` (shown in Figure 7) by postprocessing. At that rate, the only class `AbstractLineDecoration` of Cluster 8 would be merged into Cluster 1, and then

ArrowTip of Cluster 6 would be also merged to Cluster 1, because AbstractLineDecoration implements the interface defined by LineDecoration who is a member of Cluster 1 and ArrowTip extends the class AbstractLineDecoration. Instead of further presenting the results for the remaining packages in detail, all the clustering results are summarized.

Summary of clustering results. As shown in Table 2, compared with the number of initial unit clusters (i.e., NOC), the number of partitioned clusters (i.e., NNC+NUC, after postprocessing) for packages figures and standard, decreases from 35 to 7 and from 64 to 18 respectively, and the corresponding reduction ratio achieves to 80% and 71.9%, respectively. For package contrib and util, great improvements are also made. The number reduces from 46 to 21 and from 41 to 27 respectively, and the corresponding reduction ratio achieves to 54.3% and 34.1%. Nevertheless, for package framework, the number witnesses no any major changes, especially afore the postprocessing. In fact, this package is constructed merely in order to provide a high-level framework for the entire system. Almost each one of its members represents a separate side of the system and few direct design pattern relationships among them can be found.

Table 2. Summarized results for packages

Package	NOC[1]	NOD[2]	Before postprocessing			After postprocessing		
			NNC[3]	NUC[4]	RR[5]%	NNC	NUC	RR%
figures	**35**	7	5	3	**77.1**	6	1	**80.0**
standard	**64**	8	7	21	**43.8**	11	7	**71.9**
contrib	**46**	11	6	15	**54.3**	6	15	**54.3**
util	**41**	4	6	28	**17.0**	6	21	**34.1**
framework	**26**	1	1	24	**3.8**	3	16	**26.9**

[1] NOC = number of classes.
[2] NOD = number of relied DPIs.
[3] NNC = number of non-unit clusters.
[4] NUC = number of unit clusters.
[5] RR = ratio of reduced clusters, i.e., 1- (NNC + NUC) / NOC.

4.2. Comparison

This subsection compares the clustering results generated by TasteJ with those produced by Bunch [15]. Bunch is a well-known software structure clustering tool developed by Mitchell and Mancoridis [18]. It clusters program units or modules such as files and classes relying on the information contained in a module dependency file, which consists of dependency relationships among modules, for instance, function calls or inheritance relationships. Bunch version 3.3.6 provides three clustering methods, i.e., generic algorithm, hill climbing and exhaustive search. We used its default setup (hill climbing method) to respectively cluster the classes of above-mentioned JHotDraw6.0's packages. Figure 8 shows the clustering result of package figures, which is taken as an example to illustrate the difference between the approach Bunch applied and our proposed approach.

The classes of package figures have 168 dependencies between them. They are divided by the tool Bunch into four clusters: two big clusters whose center classes are PolyLineFigure and AttributeFigure, and two small clusters whose center classes are BorderDecorator and GroupFigure (NullFigure, NullConnector and UngroupCommand of package figures are not covered by any of these clusters since no dependency relationship can be found among them, or between them and the other classes within the package). The partitioned result displays expected high cohesion and low coupling.

However, we found it is still difficult for the reviewers to build a mental model for the internal classes of a certain resulting cluster, especially the big-size cluster, since Bunch solely focuses on the dependency relationship among classes, inclines to group the hotspot class and its intimate neighbors into a common group, and neglects some related design information. TasteJ provides a new perspective for clustering classes. As indicated by the cluster names in Figure 7, the partitioned clusters are able to vividly reveal their described subjects which are much easier to catch and handle, since TasteJ depends on the design pattern information that contains not only structural information on the target programs, but also the design intents left by the system developers.

4.3. Discussion

Our approach is promising to produce more understandable clustering results by bringing in design pattern relationships to direct the clustering process. It is worth noting that we did not use any lexical matching skills in the approach. Though, for certain packages, the clustering result may contain too many unit clusters, we are able to use postprocessing to effectively reduce its size without generating much impact on each cluster's original described themes. Unfortunately, our approach is not suitable for the special-purposed package that is constructed merely for providing a high-level skeleton or framework and contains few direct design pattern relationships, e.g., package framework. In brief, the clustering results on the project JHotDraw6.0 show that our proposed approach is feasible and promising in most situations by taking design pattern information into account.

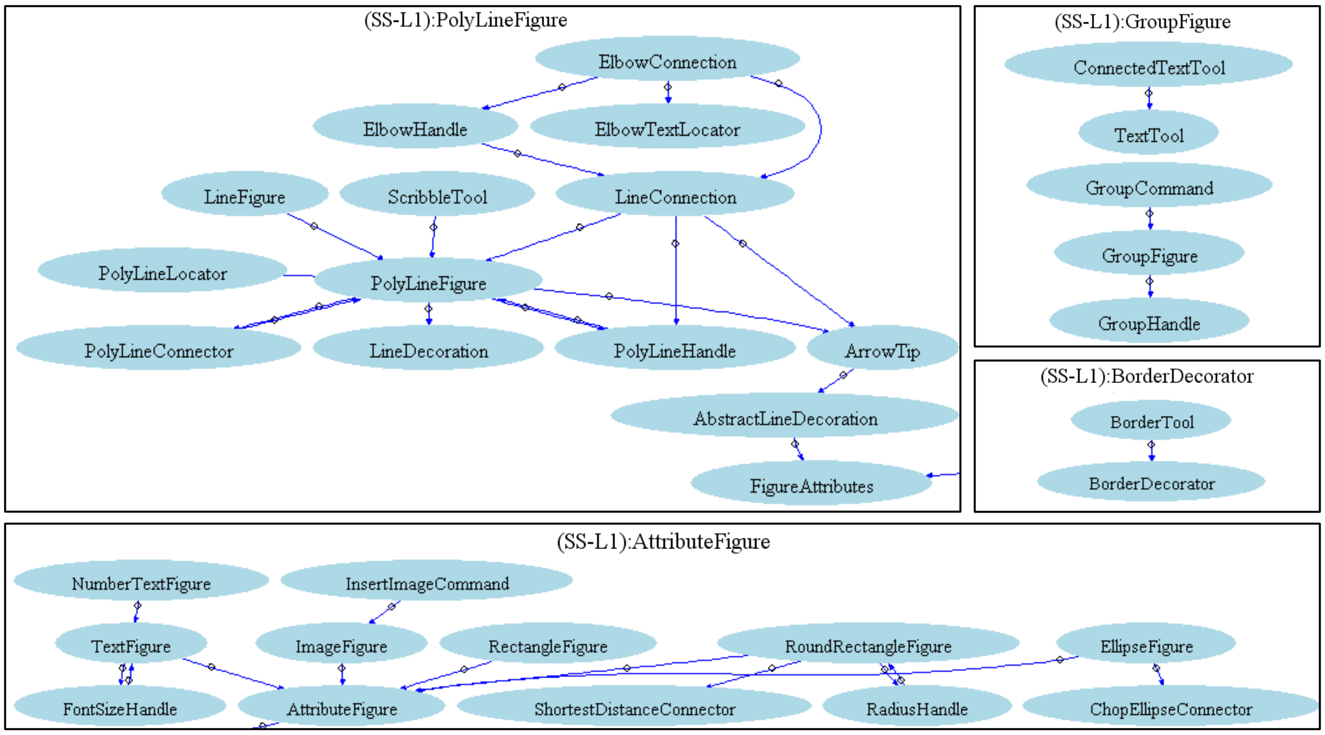


Figure 8. Partitioned clusters of package figures (generated by Bunch)

5. Related work

Many software partition and clustering approaches and tools were proposed to facilitate the understanding of a software system. The above-mentioned tool Bunch [18, 16] decomposed a system by partitioning a module dependency graph. Mahdavi et al. [13] treated subsystem decomposition as a search problem and tried to solve it using a multiple hill climbing approach. Saeed et al. [20] proposed a combined clustering approach for grouping software entities. Maletic et al. [14] and Kuhn et al. [11] clustered software artifacts using latent semantic indexing, an information retrieval technique, based on the assumption that parts of the system that use similar terms are related. Beyer and Noack [2] used file co-change information that extracted from the version control repository to identify candidate clusters. Andritsos and Tzerpos [1] clustered a software system based on minimizing information loss during the software clustering process.

Distinct from these partition and clustering approaches, the clustered object of our approach can be part of a system, e.g., a package, and an entire system is not always needed; moreover, the clustering approach we designed groups classes not according to the traditional coupling relationship, but the design pattern relationships among roles; furthermore, we did not bring in any complex computing techniques.

In addition, Tzerpos and Holt [27] proposed a distance metric *MoJo* to compare two decompositions. Mitchell and Mancoridis [17] used similarity measurements to evaluate different decompositions. Harman et al. [8] studied the robustness of clustering fitness functions.

Many visualization tools were also proposed to assist program comprehension. CodeCrawler [12] used polymetric views to show software entities and metrics. SHriMP [24] used a nested graph view to provide multiple perspectives of information at different levels of abstraction. CodeCity [30] used 3D visualization technique to describe system contents through treating a code system as a virtual *city*. The tool presented by literature [31] allows programmers to reverse engineer LePUS3 charts from Java 1.4 programs at any level of abstraction.

Our tool TasteJ is not merely a tool for visualizing clustering results. It aims at providing a new perspective for understanding source code. It displays the internal structure of a cluster using a refined class diagram and indicates the classes' acted roles when needed. Moreover, it provides separate views for inspecting the internal details of DPIs.

6. Conclusion

We have proposed a novel design pattern directed clustering approach to assist the understanding of open source code. A prototype tool TasteJ has been developed to facilitate understanding Java programs. Moreover, we conducted a case study on an open source system JHotDraw6.0, and compared TasteJ with Bunch, a state of art software structure clustering tool. The preliminary result shows that the approach is feasible and promising. So far, TasteJ is not suitable for the situation that some classes are put together merely for building a high-level skeleton or framework. And it might be weak for the programs none of design patterns is considered during their development. In the future, we will evaluate our approach with more open source projects, and improve its scalability, extensibility and flexibility.

Acknowledgements. We gratefully thank Xue Zhang for suggestions on the language expression in this paper.

References

[1] P. Andritsos and V. Tzerpos. Software clustering based on information loss minimization. In *Proc. of 10th WCRE*, pages 334–344.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proc. of 13th IWPC*, pages 259–268, 2005.

[3] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. Design pattern recovery by visual language parsing. In *Proc. of 9th CSMR*, pages 102–111, 2005.

[4] J. Dong, D. Lad, and Y. Zhao. DP-Miner: Design pattern discovery using matrix. In *Proc. of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*, 2007.

[5] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures*, 30(1-2):21–33, 2004.

[6] R. Ferenc, A. Beszedes, L. Fulop, and J. Lele. Design pattern mining enhanced by machine learning. In *Proc. of 21st ICSM*, pages 295–304, 2005.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements Of Reusable Object-Oriented Software*. Addison-Wesley Reading, MA, 1995.

[8] M. Harman, S. Swift, and K. Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1029–1036. ACM New York, NY, USA, 2005.

[9] JHotDraw6.0. http://www.jhotdraw.org.

[10] O. Kaczor, Y. Gueheneuc, and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. In *Proc. of 10th CSMR*, volume 6, pages 175–184, 2006.

[11] A. Kuhn, S. Ducasse, and T. Gîrba. Enriching reverse engineering with semantic clustering. In *Proc. of 12th WCRE*, 2005.

[12] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, pages 782–795, 2003.

[13] K. Mahdavi, M. Harman, and R. Hierons. A multiple hill climbing approach to software module clustering. In *Proc. of 19th ICSM*, pages 315–324, 2003.

[14] J. Maletic and N. Valluri. Automatic software clustering via latent semantic analysis. In *Proc. of 14th ASE*, pages 251–254, 1999.

[15] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner. Bunch: a clustering tool for the recovery and maintenance of software system structures. In *Proc. of 13th ICSM*, pages 50–59, 1999.

[16] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen, and E. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. of 6th IWPC*, pages 45–52, 1998.

[17] B. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measurements. In *Proc. of 17th ICSM*, pages 744–753, 2001.

[18] B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, pages 193–208, 2006.

[19] J. Niere, W. Schafer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of 24th ICSE*, volume 19, pages 338–348, 2002.

[20] M. Saeed, O. Maqbool, H. Babri, S. Hassan, and S. Sarwar. Software clustering techniques and the use of combined algorithm. In *Proc. of 7th CSMR*, pages 301–306, 2003.

[21] N. Shi and R. Olsson. Reverse engineering of design patterns from java source code. In *Proc. of 21st ASE*, volume 18, pages 123–134, 2006.

[22] J. Smith and D. Stotts. SPQR: Flexible automated design pattern extraction from source code. In *Proc. of 18th ASE*, pages 215–224, 2003.

[23] M. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Proc. of 13th IWPC*, pages 181–191, 2005.

[24] M. Storey and H. Müller. Manipulating and documenting software structures using SHriMP views. In *ICSM*, pages 275–284, 1995.

[25] TasteJ. http://seg.nju.edu.cn/tastej.

[26] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, pages 896–909, 2006.

[27] V. Tzerpos and R. Holt. MoJo: A distance metric for software clusterings. In *Proc. of 6th WCRE*, pages 187–193. IEEE Computer Society Washington, DC, USA, 1999.

[28] M. Vokáč. An efficient tool for recovering Design Patterns from C++ Code. *Journal of Object Technology*, 5(1):139–157, 2006.

[29] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA)*, 2003.

[30] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proc. of 15th ICPC*, pages 231–240, 2007.

[31] R. Wettel and M. Lanza. The design navigator: charting java programs. In *Proc. of 30th ICSE*, pages 945–946, 2008.