Tracking Objects to Detect Feature Dependencies

Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz

Software Composition Group
University of Berne, Switzerland
{lienhard, greevy, oscar}@iam.unibe.ch

Abstract

The domain-specific ontology of a software system includes a set of features and their relationships. While the problem of locating features in object-oriented programs has been widely studied, runtime dependencies between features are less well understood. Features cannot be understood in isolation, since their behavior often depends on objects created and referenced in previously exercised features. It is difficult to spot runtime dependencies between features just by browsing source code. Hence, code modifications intended for one feature, often inadvertently affect other features. In this paper, we propose an approach to precisely identify dependencies between features based on a fine-grained dynamic analysis which captures details about how objects are referenced at runtime. The results of two case studies indicate that our approach helps software maintainers in understanding critical feature dependencies.

Keywords: reverse engineering, dynamic analysis, feature analysis, object aliasing, visualization

1 Introduction

A feature is a unit of domain functionality as understood from the user's perspective. During requirements analysis, relationships between features are specified to express conceptual dependencies and constraints of a system [21]. Correct specification of dependencies is vital to ensure correct behavior of a system and to avoid behavioral problems.

Much of the feature-related research for system comprehension focuses on *feature identification*, a technique for locating parts of code that implement features [25, 26, 7, 1]. Only few researchers have investigated relationships between features [22, 9, 14].

The runtime behavior of object-oriented systems is characterized by object instantiations and message sends. Objects may be long-lived and used by many different features of a system. Before a feature can be exercised, it may require other features to establish a particular program state. In a previous work, we defined relationships between features based on shared usage of static entities (*i.e.*, classes and methods) [9]. A static perspective however, overlooks runtime characteristics of object-oriented systems. Our focus in this work is on analyzing runtime feature dependencies. We consider a runtime dependency to exist between features if state changes in one feature potentially impact the behavior of another feature.

Salah *et al.* described a technique to identify runtime dependencies between features by detecting situations where objects are created in one feature and are later used in another feature [22]. However, considering only object instantiation is not sufficient to detect all runtime dependencies. We also need to consider *object aliasing*, a situation which occurs when multiple references to an object exist [12]. The approach of Salah *et al.* only considers object creation, thus it misses dependencies between features that result from one feature accessing object aliases (*i.e.*, references to objects) in features other than the one where an object was originally instantiated.

In this paper, we propose a technique that tracks objects at runtime to detect dependencies between features. As a motivation for our work, we address the following research questions:

1. *What is an accurate definition of feature runtime dependencies?* In the context of object-oriented programming we need to define a notion of dependencies which also considers object aliasing.

2. *Which runtime dependencies exist between features?* Understanding runtime dependencies between features is essential for carrying out maintenance tasks without inadvertently breaking seemingly unrelated code.

3. *How can we interpret feature dependencies?* To capture feature dependencies, we need to determine on which objects a feature depends. For example, changes

15th IEEE International Conference on Program Comprehension (ICPC'07)
0-7695-2860-0/07 $20.00

in the implementation of one feature may affect the object state used by another feature. A challenge of analyzing runtime dependencies between features at the level of objects is that it is difficult to attribute semantic or contextual interpretation to these dependencies.

To tackle these questions, we propose an approach that adopts a fine-grained dynamic analysis technique which traces the transfer of object references at runtime. We refer to our technique as *Object Flow Analysis* [16].

The key contribution of this paper is a new definition of runtime dependencies and a detection strategy based on a meta-model which captures object aliases. The new definition is shown to be more precise than the one proposed by Salah *et al.* [22]. Furthermore, to help a software engineer interpret dependencies, we propose a visualization which supports understanding the relationships between objects a feature depends on.

Paper structure. In the next section we discuss problems of aliasing in object-oriented programs that complicate the detection and comprehension of runtime dependencies between features. In Section 3 we briefly introduce object flow analysis, as it serves as a basis for our fine-grained dynamic analysis approach. Subsequently, in Section 4 we introduce our approach. In Section 5 we apply our approach to two case studies and detail the results we obtained. We discuss different aspects of our approach in Section 6. Section 7 outlines related work and we conclude in Section 8.

2 The Problem of Feature Dependencies

Functional requirements are often centered around features since they reflect the end-user's perspective of a system. We adopt the definition of a feature proposed by Eisenbarth *et al.*: *"A feature is a realized functional requirement of a system. A feature is an observable unit of behavior of a system triggered by the user"* [6].

Features and their relationships are not represented explicitly in the source code of object-oriented systems. However, a software engineer frequently needs to understand which parts of a system implement a feature to carry out maintenance activities, as change requests and bug reports are usually expressed in terms of features [18]. Furthermore, a software engineer needs to understand relationships between features, as code modifications to one feature may cause unexpected side effects in other features.

2.1 Runtime Dependencies Between Features

The behavior of one feature may depend on certain program state being established during the exercising of another feature. For example, in a Mail Client application, a "send mail" feature may require a "compose mail" feature to set mail recipients before it can be exercised.

During program execution, the object reference graph (*i.e.*, the objects and their reference relationships) steadily changes, as new objects are created, references between objects are changed, or objects are garbage collected. As a feature is exercised, it typically produces side effects on this graph. Therefore, since program behavior depends on the reference relationships of objects, the behavior of a feature may be influenced by a previously exercised feature.

To analyze those dependency situations we have to consider the interrelationships between objects. This is complicated by the fact that there may exist multiple access paths to the same object – a situation generally referred to as *object aliasing*.

Object aliasing is endemic in object-oriented programming and has been recognized to cause problems like representation exposure [17] or argument dependency [19]. While strategies to prevent aliasing problems at compile time have been more widely studied [12], the dependencies caused by aliasing at runtime are less well understood.

2.2 Why Object Aliases Cause Dependencies

Intuitively, we consider a feature to depend on another one if object state is changed in the first feature and then the second feature's behavior uses this state.

Salah *et al.* [22] define a relationship $depends$ from a feature F_i to a feature F_j if F_i uses objects that are created by F_j (*i.e.*, F_i *depends on* F_j). Let $I(F)$ be the set of objects used by F (*i.e.*, the objects imported by a feature), and $E(F)$ be the set of objects created by F (*i.e.*, the objects that can be exported by a feature), then:

$$depends \equiv \{(F_i, F_j) \mid I(F_i) \cap E(F_j) \neq \emptyset, i \neq j\}$$

To *use* an object in this context means that the object is sent a message (not including changing references to it). This definition, however, does not capture all runtime dependencies. Let's consider the following simplified case illustrated by Figure 1. It shows the features *Startup*, *Join Channel*, and *Receive Message* of an IRC chat client from one of our case studies presented in Section 5.

Between each feature, we show the snapshot of live objects; the first taken before running the *Join Channel* feature and the second before the *Receive Message* feature. We treat a snapshot as a directed graph, commonly termed *object reference graph*. Nodes represent objects and edges represent a field of one object referring to another object.

While exercising the feature *Startup*, two objects, a window object (`w`) and connection object (`c`), are created. Then, while exercising the feature *Join Channel*, the first snapshot is transformed into the second. It creates an observer object (`o`) and assigns the connection object to one of its fields, *i.e.*, a reference $o \rightarrow c$ is created. Now the object c is aliased since there are two objects referring to it.

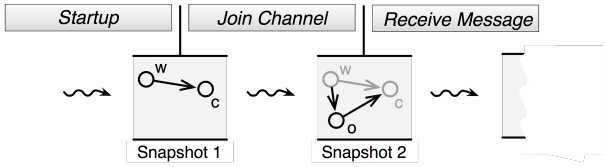

Figure 1. State changes between features.

Let's assume that in the *Receive Message* feature, the observer sends messages to the connection through the object reference $o \rightarrow c$. Therefore, the *Receive Message* feature depends on the *Join Channel* feature. The rationale is that without the appropriate state changes in *Join Channel*, *Receive Message* would exhibit a different behavior, or in the worst case, abort with a null pointer exception.

The $depends$ relationship proposed by Salah *et al.* [22] is not capable of detecting that *Receive Message* depends on *Join Channel*. It only detects the dependency on the *Startup* feature, in which the connection is instantiated.

We conclude that a dependency detection strategy as described by *depends* at the object level (object creation and usage) is not precise enough to detect this type of dependency. We solve this problem by tracking object aliases.

3 Object Flow Analysis in a Nutshell

Typically, dynamic analysis approaches trace execution of a system by capturing the sequence and nesting in which methods are executed [3, 7, 14]. In contrast, our *Object Flow Analysis* adopts an orthogonal view of runtime behavior, by capturing how object references are passed around the system at runtime. An *Object Flow* represents the life cycle of an object at runtime, *i.e.*, where it was instantiated and how it was then passed through the system.

To trace the flow of objects, we exercise features on an instrumented system and capture both the message events (referred to as *activations)* and the object flow data (*i.e.*, the creation of object aliases). Our meta-model treats object aliases explicitly [16]. We consider an object alias to be created when an object is (1) instantiated, (2) stored in a field (*i.e.*, instance variable) or global, (3) read from a field or global, (4) stored in a local variable, (5) passed as argument, or (6) returned from a method execution.

The rationale is that each object alias is bound to exactly one activation, namely the activation in which the alias makes the object visible. By definition, arguments, return values, and local variables are only visible in one method activation. In contrast, objects that are stored in fields (*i.e.*, instance variables) or globals, can be accessed in other activations as well. Therefore, we distinguish *read* and *write* access of fields and global variables.

Apart from the very first alias, which stems from the object instantiation primitive, all aliases are created from an existing one. This gives rise to a *parent-child* relationship between aliases originating from the same object.

4 A Detection Strategy for Fine-grained Dependencies

As discussed in Section 2, a precise notion of runtime dependencies between features needs to distinguish the references through which messages are sent to an object.

We define the runtime dependency relationship $depends_{new}(F_i, F_j)$ to show that F_i depends on F_j. Let $C(F)$ be the set of aliases created in F and $U(F)$ be the set of aliases used in F to send messages to objects. Also, let $A(a)$ be the transitive closure of the *parent* relationship (*i.e.*, ancestors of alias a), extended to sets of aliases, then:

$$depends_{new} \equiv \{(F_i, F_j) \mid A(U(F_i)) \cap C(F_j) \neq \emptyset, i \neq j\}$$

In other words, a feature depends on another if any of the objects it uses can be traced back as originating from the latter feature. This definition represents our detection strategy and yields a superset of dependencies compared to $depends$ as defined in Section 2.2. For each object created in a feature, there exists an alias which is the root of all subsequently created aliases. Therefore, an object that is subject to a dependency, $depends_{new}$ includes the feature in which the object was created.

Our new definition detects additional dependencies caused by aliases created in features other than the one in which the object was instantiated.

4.1 Detecting Runtime Dependencies

We now discuss how our analysis detects runtime dependencies as specified above. We defined a object-flow meta-model to express features as a collection of object aliases, activations and their relationships. We implemented a tracing tool (Object Flow Tracer) to track aliases and method activations of features.

In our object flow meta-model, illustrated by Figure 3, we identify the alias through which a message is sent to an object (the receiver of an activation is an Alias rather than an Instance). Each alias knows the method activation in which it was *created* and which made the instance visible. Aliases and activations each have a link to their parent (*i.e.*, the sender instance of an activation). For conciseness, Figure 3 omits the different types of aliases and how they are linked with the static model. For more details of our meta-model, we refer the reader to our previous work [16].

Based on our meta-model, we detect runtime dependencies for a given feature F. First we find all aliases through

```
context Alias
  def allParents: Set(Alias) =
    self.parent->union(self.parent->collect(p|p.allParents))

context Feature
  def dependencies: Set(Feature) =
    self.activation.receiver.allParents.creator.feature->reject(self)
```

Figure 2. OCL specification of a feature's dependencies.

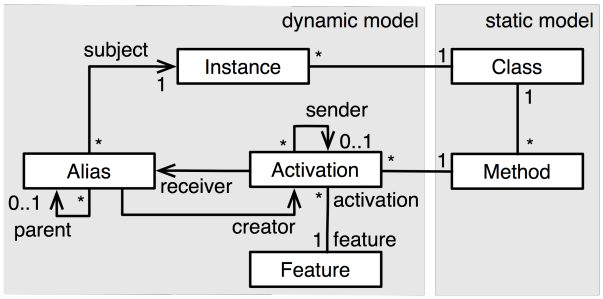

Figure 3. Core of the object flow meta-model.

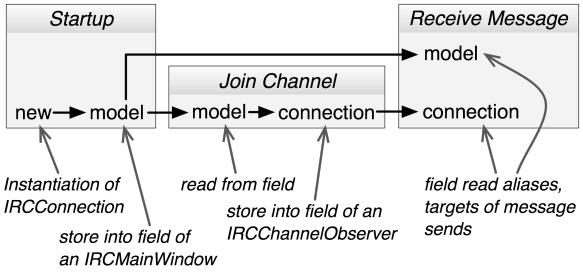

Figure 4. Object flow of an IRCConnection.

which messages are sent in F, *i.e.*, the receivers of the activations of F. Then, the parent chain of each detected object alias is traversed backward. If a parent alias originates in a feature other than the current feature F, we consider that we have detected a dependency.

We define the dependencies of a feature in OCL as shown in Figure 2. `allParents` yields the transitive closure of the parents of an alias. This is then used to navigate from a feature to all features (excluding itself) on which it depends. For a feature, the resulting features of the query `dependencies` satisfy the $depends_{new}$ relation.

4.2 Example

Figure 4 illustrates an example of an Object Flow from our case study, IRC chat client. It shows the flow of an IRCConnection instance between the features *Startup*, *Join Channel*, and *Receive Message*.

Let's assume that we look at the *Receive Message* feature to find its runtime dependencies. The connection instance is a candidate, as it receives messages while the feature is exercised. The aliases through which the messages are sent are two field read aliases. The first is from the field named model in a main window instance. The second is from the field named connection in a channel observer.

To determine whether the *Receive Message* feature has a dependency on other features with respect to the connection instance, we trace back its flow. The parent of the model field read alias is in the feature *Startup*. On the other hand, the connection field read alias first leads back to *Join Channel*, and further back in the flow, it joins the other flow in *Startup*. Therefore, we detect that *Receive Message* depends on both *Join Channel* and *Startup* features.

In Section 2.2 we showed that object aliasing potentially introduces dependencies on features even if the objects depended on were created in a different feature (see Figure 1). Figure 4 illustrates the same situation and we discuss how our approach also detects these dependency situations.

In the remainder of this section, we discuss how we support a software engineer to understand runtime feature dependencies by providing information about how they are related to each other.

4.3 Exploring Object Dependencies

With our detection strategy, we obtain for a feature under investigation, a set of other features it depends on. Each dependency points out an object alias that was stored in a field during the exercising of a previous feature. Through this field, the object is then accessed and used in the feature. If the field was assigned with a different object, or not assigned at all, the behavior of the feature would likely be altered or the execution would result in an error.

To help understanding the details of a dependency, as it is required when carrying out maintenance work, the objects depended on in the feature need to be made explicit.

The challenge we face is that there are potentially hundreds of objects on which a feature depends. Thus, it is difficult to understand the dependencies in isolation. What is missing are the relationships between objects. If we view object dependencies in a larger context we can interpret them more easily and attribute semantic meaning to them.

An interesting observation we made is that most objects depended on are part of a reference relationship to another object dependency. This is plausible because, to use an object, a feature often needs to access another object which stores a reference to it. Consequently, the feature also depends on the object from which the reference was accessed. The exceptions are objects referred to from outside the application, *e.g.*, from the UI framework or the main method.

Based on these reference relationships of the object dependencies, we build the *object dependency graph*. Figure 5 illustrates such a graph taken from the *Receive Message* feature of the IRC chat client case study. Each node represents an object depended on in the feature. Each edge represents a reference that is subject to a dependency, *i.e.*, the reference is accessed in the feature but created in another one. In other words, the object dependencies of a feature directly map to the references (edges) shown in the graph.

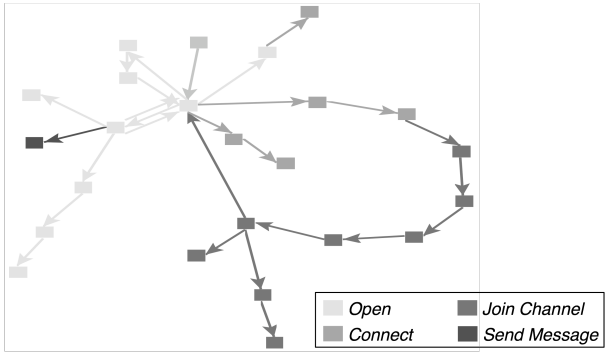

Figure 5. Object dependency graph of Receive Message feature.

We use grayscale to convey information about which feature an object or a reference was created in. Light gray means an object or reference was created in a feature that was exercised early in the run; dark gray in a more recent one. We apply a force-based layout algorithm to visualize the graph.

We integrated this visualization into our Object Flow Tracer tool used to carry out the case studies. In this tool, the class name of an object is shown in a tooltip when moving the mouse over it. Accordingly, the tooltip of a reference shows in which feature it was created.

In the IRC Chat Client case study presented in the subsequent section, we discuss the object dependency graph illustrated by Figure 5 in more detail. We discuss how this view helped us to understand object dependencies in the context of exercising a feature. This is key to be able to attribute semantical meaning to feature dependencies.

5 Validation

We structure the discussion of the case studies based on two questions presented in the introduction.

To answer the first question, "*Which runtime dependencies exist between features?*", we compare our approach to the one of Salah *et al.* [22]. To do so, we implemented their approach and compared the resulting dependencies with ours. Furthermore, we focussed on the additionally detected dependencies. We were interested to find out how important those kind of dependencies are and which role they play among the other dependencies of a feature.

To answer the second question, "*How can we interpret feature dependencies?*", we used the object dependency graph to evaluate how helpful it is to support understanding the relationships between the dependencies.

5.1 Case Study: IRC Chat Client

As a first case study we chose an IRC Chat Client [24]. It is implemented in Squeak, an open source dialect of Smalltalk [23]. Our motivation was (1) because it is a small (39 classes and 1063 methods) but non-trivial legacy application and (2) we have access to the source code. A total of six developers contributed to the project which underwent various refactorings and enhancements over nine years.

Which runtime dependencies exist between features? We exercised nine distinct features. Figure 6 shows the features in the order they were run from top to bottom with the number of dependencies they have. The feature *Startup* naturally does not have dependencies because it is run first. In all subsequent features our analysis found dependencies upon previous features.

Figure 6 also provides the number of dependencies of Salah's approach. It shows that compared to their approach, we detect more dependencies in the *Connect* feature and all subsequent features. In the second feature, *Setup*, we found exactly the same number of dependencies. This is what we expected as this feature was the second feature we exercised, it cannot depend on more than one feature.

We now present some anecdotal evidence indicating that the additionally found dependencies play a central role among the other dependencies of a feature for maintenance.

For instance, the *Connect* feature depends on both the *Open* and the *Setup* feature with respect to the connection instance. The connection was created in *Open*, hence, Salah's approach only finds this dependency.

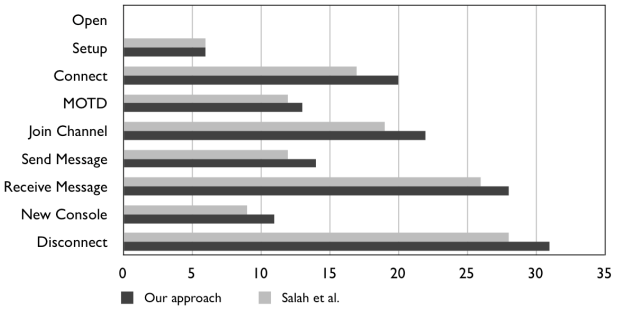

Figure 6. IRC features and dependencies.

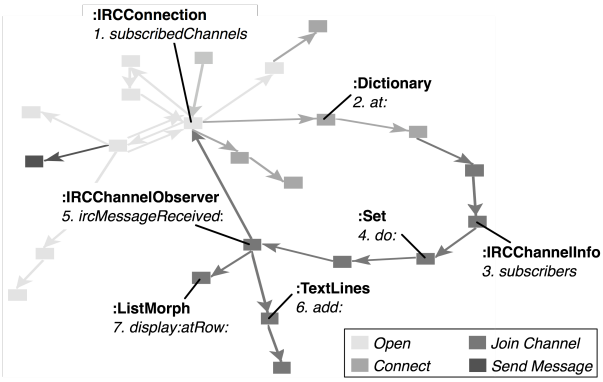

Figure 7. Object dependency graph of Receive Message feature annotated with invoked methods.

A closer investigation showed that the setup dialog created in the *Setup* feature was still alive and had a reference to the connection to poll its state. This coincides with the documentation which states that it disables user input when connected. To our surprise, however, this dependency occurred in the *Connect* feature, although the dialog had been closed earlier. The conclusion we can draw from this is that closing the setup window did not clean up correctly and hence left instances behind which continued to be used.

The same dependency also occurred in all subsequent features and hence contributed to the number of additional dependencies.

Another additional dependency is the one discussed as an example in Section 4.2 (see Figure 4). It illustrates how *Receive Message* depends not only on *Startup* but also on *Join Channel*. We found a very similar dependency on *Join Channel* in the feature *Send Message*. This reflects a domain constraint: sending and receiving messages requires to take place in an IRC channel.

How can we interpret feature dependencies? To support the interpretation of runtime dependencies of a feature, we used the object dependency graph visualization.

As a concrete example, we discuss the dependencies of the *Receive Message* feature. We already illustrated its object dependency graph in Section 4.3. Figure 7 presents a part of this graph with annotations of instances and messages sent to them in the feature (in our tool we can access this information interactively on demand).

Striking is the long loop starting at the IRCConnection object. The connection holds a dictionary which maps channel names to a channels (instances of IRCChannelInfo). We can see that the channel was not created in the *Connect* feature, but later in the *Join Channel* feature. It holds a channel observer stored in a set.

The object dependency graph reveals the composition hierarchy of objects depended on in the context of the feature. Inspecting the message sends further helps us to map the object dependencies to the runtime behavior of the feature.

Based on our analysis, we extract and reconstruct the following activity in the *Receive Message* feature. First, the connection gets the appropriate channel for the message received (see messages subscribedChannels and at:). Then the channel iterates over the set of subscribers to notify them of a received message (ircMessageReceived:). The observer adds the new message to its text lines object which is the model of the UI list widget being updated with display:atRow:.

Summarizing, the object dependency graph reveals the following key information about the runtime dependencies of a feature:

- The software engineer quickly grasps how many and which object dependencies originate in a feature, and whether this feature was exercised recently or earlier in the program execution. In Figure 7 we see that most objects are created in three stages, namely in the *Open*, *Connect* and *Join Channel* features.

- For each object dependency, the incoming references show through which other object(s) it was accessed while the feature was exercised. The brightness of a reference indicates in which feature the reference was created. In Figure 7, for example the list morph and text lines objects are only accessed through the channel observer, whereas the connection received messages from multiple objects.

- It shows object dependencies that served as starting points to further extensions of the object graph in a later run feature. An example is the connection which stores a dictionary of channels. This dictionary is not created in the same feature as the connection but later in the *Connect* feature.

- It shows object aliasing, *i.e.*, an object referred to by more than one other object. The software engineer can spot aliases of an object that are created in a later feature than the one which created the object. For example the connection instance is aliased. We can see that some aliases were created in the same feature (*Open*) as the connection itself. Two aliases to the connection, on the other hand, were created in later features.

The aliasing situation described in the last point is particularly interesting because it allows one to visually spot dependencies on objects referenced in one feature but created in another. These are exactly those cases which our approach is capable of detecting because it does not only consider object creation and usage but also tracks aliases.

Summary of results. The case study showed that Salah *et al.* indeed captures most of the feature dependencies. However, the additional dependencies that we uncover are precisely the indirect feature dependencies that can be problematic during maintenance. In one case, a dependency even pointed out an anomaly of the program.

The object dependency graph visualization proved to be of great help for our analysis – it was much simpler than if we had looked at the dependencies one by one. The visualization allowed us to get a quick overview of the dependencies of a feature but also helped us to spot interesting dependency situations.

5.2 Case Study: Pier

Pier is a web content management system [20]. It is a reengineered version of SmallWiki [5] ported to Squeak [23]. Its core comprises 177 classes and the meta-model 200 classes. Our choice of Pier was motivated by the following reasons: (1) it is open source, thus its source code is freely available, (2) we are familiar with the predecessor application SmallWiki and have also analyzed this application in previous research [9, 10], (3) we are familiar with the features of Pier from the user's perspective, and (4) we have direct access to developer knowledge to verify our findings.

Which runtime dependencies exist between features? For our experiment, we traced 11 features as listed in Figure 8. The average number of dependencies per feature is much higher compared to the IRC Client cases study. Again the first feature does not have any dependencies. The comparison with the approach of Salah *et al.* shows that we found additional dependencies in all features, except for the first two (for the same reason as explained above).

All features are related to *Initialization*, the feature that is exercised to load the Pier application, and to the *Start* feature which displays the first web page. This is because these features are responsible for initializing the system, the user session, and its UI components.

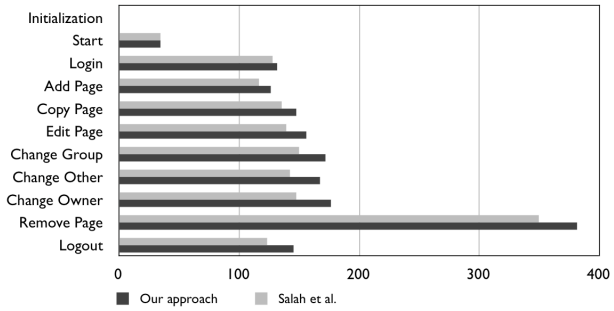

Figure 8. Pier features and dependencies.

As with the IRC Client case study we analyzed the additionally detected dependencies. Again, it turned out that they play important roles in the system.

For example, in all features we see a dependency on an instance of User. It is created during the feature *Initialization*. In the *Login* feature (we logged in as administrator) it is accessed from the kernel and stored in a context object. In each subsequent feature this user object is then used for controlling access. Additionally, in the feature *Change Owner*, we see that the user is accessed also from the page on which we are changing the owner. The object dependency graph accurately shows that the reference from the page to the owner was created in the feature *Add Page*. This means, that when the page was instantiated, the user creating it is assigned to be its owner.

How can we interpret feature dependencies? By exploring the object dependency graphs we notice that some of the dependencies are recurring (*i.e.*, the same dependencies exist in most of the features). The dependencies are due to the nature of the Pier application as a web application. Thus every feature makes use of page rendering activity. For example, the *Login* feature reveals similar dependencies to the *Start Page*. This is due to the page being redisplayed after the login action has completed. This characteristic of Pier explains the much higher average number of dependencies compared to the IRC Client case study.

A revealing observation was that the object dependency graph of most features reflects the hierarchical structure of pages. We see that Pier has a very fine-grained object model to represent content (*e.g.*, lists are composed from list items, each containing text or link objects etc.). Behavior like page rendering or copying is performed by Visitors which traverse the full object trees. Hence, there exist a large number of dependencies on the features that create or copy pages.

The order of exercising the features also impacts the dependencies. For example, each feature accesses a PRContext instance created by the previously exercised feature.

The *Remove Page* feature stands out in Figure 8 with a much larger number of dependencies compared to the other

Figure 9. Object dependency graph of the Remove Page feature from Pier.

features. This surprised us because we expected that removing a page would be a rather small feature.

Figure 9 illustrates the object dependency graph of the *Remove Page* feature. It contains large trees representing all existing pages of Pier. There are three default pages at the left side and two smaller pages at the right. Latter are created by *Add Page* and *Copy Page*. A closer investigation showed that when the Pier application removes a page, it iterates over the entire structure to check for the existence of links to the page that is to be removed. This activity generates a lot of dependencies.

Summary of results. The Pier application showed very different characteristics compared to the IRC Client. It comprises a larger number of dependencies. There are two reasons. (1) being a web application the web page is regenerated on each request, and (2) the domain model of Pier is larger and much more fine-grained.

As in the IRC Client case study our approach detected additional and invaluable dependencies which are missed with the approach of Salah *et al.*

6 Discussion

In our evaluation we do not consider precision of our approach because the analysis tool implements the meta-model and detection strategy as specified in Section 4. Considering recall, a noteworthy limitation of our approach is the well-known fact that dynamic analysis is not exhaustive, as all possible paths of execution are not exercised [2]. Therefore, an analysis of feature runtime dependencies always has to be understood in the context of the actual execution. While this is a difficulty, at the same time it is a key characteristic as running a feature can be related directly to internal program behavior.

To cover all relevant dependency situations in a concrete program execution, the tracing technique has to be consider carefully. Our Object Flow Tracer not only traces objects of application classes but also instances of system classes. For example, collections and arrays have to be taken into account because they preserve permanent object references between the holder of the collection and its contained elements. Furthermore, since all messages sent to an object have to be traced, also the class Object has to be instrumented.

Language independence. To perform object-flow analysis, we need to capture details in the trace that reveal the path of objects through the system. We chose Smalltalk to implement our Object Flow Tracer because of its openness and reflective capabilities which allowed us to evaluate different alias tracking techniques. We believe that an implementation of the Object Flow Tracer for example in Java is possible. A potential problematic area is the instrumentation of system classes or the tracking of primitive object types. The meta-model (both the static and dynamic part) and the detection strategy we describe are language independent.

Points of variation. The object dependency graph shows all objects that a feature depends on. This means that also implementation details are visible, for example the objects from which a dictionary is built. Similarly, domain model objects often encapsulate other objects that serve as their internal representations. Ideally, in a system those internal objects are strictly encapsulated, *i.e.*, all access paths go through the owner object. An enhanced visualization approach could detect such ownership situations and hide the internal objects. This would reduce information that is not strictly required to understand the dependencies.

Our approach to visualize the dependencies of a selected feature provides a perspective of looking back into the past, *i.e.*, showing dependencies on previously exercised features. Alternatively, a perspective which would provide a look ahead from the point of view of a feature could provide additional insights. Such a view would show which dependencies originate in a feature and how they may affect features exercised at a later point in time.

Scalability. From a scalability point of view, time and space complexity of the detection algorithm is linear to the number of messages sent in a feature. Rather a limiting factor is the object dependency graph visualization. It is limited in that (i) with increasing number of dependencies the visualization naturally gets harder to understand and (ii) with too many different dependent features (about 10 or more), the different grayscale colors are not easily distinguishable anymore.

7 Related Work

Our work is directly related to the fields of feature related research [2, 6, 18, 9] and dynamic analysis [2], which covers a number of techniques for analyzing information gathered while running the program, and visualization techniques for presenting object-oriented program behavior.

Our work builds on the analysis of runtime feature relationships, pioneered by Salah and Mancoridis [22]. Their approach built on well-established dynamic analysis *Feature Identification* techniques (*e.g.*, *Software Reconnaissance*) [25] to extract features. They defined a hierarchy of dynamic views which track inter-feature dependencies. As discussed in Section 2.2, their main definition $depends$ detects situations where an object is used in a different feature than the one it is created in. We show why this definition misses crucial dependencies and we propose a more precise notion of feature runtime dependencies.

Kothari *et al.* [14] proposed an approach to system comprehension that considers features as the primary unit of analysis. They define a relationship between features based on comparing the implementations of two features in terms of the executed methods. Also other approaches are based on an analysis of the executed methods, *e.g.*, for locating features in the source code [7].

Those approaches analyze the dynamic behavior of a system at the granularity of methods. For our problem, however, tracing method executions alone is not enough. The reason is that a feature potentially has a dependency relationship on another feature even without executing the same methods. Our approach detects the dependency which occurs when a feature stores an object in a field and a later run feature accesses this field in a different method.

Various approaches have extended method tracing to improve object-oriented program understanding. For example, apart from Salah *et al.* [22] mentioned above, also Antoniol *et al.* [1] consider instance creation events to locate features. In contrast, our Object Flow Analysis is much more radical as it proposes a new model which is centered around objects, capturing object aliasing, a key characteristic of object-orientation.

Related to the analysis of object references are query based approaches, which let the programmer test relationships between objects [8]. In contrast to our approach, the query based approaches are more suited to finding inconsistencies in object graphs than detecting feature dependencies. Also, *a priori* knowledge about the implementation is required to be able to write queries whereas our approach can be used to study an unfamiliar system.

To present method traces or instance creation events, many approaches exploit visualization techniques [3, 13, 15, 10]. Only few approaches visualize object reference relationships. Super-Jinsight visualizes object reference patterns to detect memory leaks [4], and the visualizations of ownership-trees proposed by Hill *et al.* show the encapsulation structure of objects [11]. Both approaches are mainly concerned with a compact representation of large object reference graphs. In contrast, for our approach, this is less of an issue because the set of objects a feature depends on is typically a relatively small subset of the complete object reference graph. Our approach provides a focused view by only showing those objects and references relevant for a feature under investigation. A novel concept of our approach is the notion of time encoded in a grayscale scheme which reveals in which feature an object or a reference was created.

8 Conclusions

In this paper we analyze the problem of runtime dependencies between features in an object-oriented system. Our work contributes to the state-of-the-art by proposing a precise feature runtime dependency definition and a detection strategy based on a meta-model which explicitly represents object references.

Furthermore, we present a dedicated visualization of feature runtime dependencies to support a software engineer in his maintenance tasks. As our case studies indicate, the visualization approach is helpful and effective for a feature analysis that requires a detailled understanding of the internals of an application.

In the future we plan to extend our approach in the following ways. We want to extend our visualization with additional interactive features, *e.g.*, to collapse or expand an owner object and its internal representation objects. Furthermore, we plan to integrate our tool into the source code browser to support exploring object dependencies directly from where they origin or where they are used in the code.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008). We also thank Tudor Gîrba for his constructive comments on this work.

References

[1] Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.

[2] Thomas Ball. The concept of dynamic analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.

[3] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[4] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.

[5] Stéphane Ducasse, Lukas Renggli, and Roel Wuyts. Smallwiki—a meta-described collaborative content management system. In *Proceedings ACM International Symposium on Wikis (WikiSym'05)*, pages 75–82, New York, NY, USA, 2005. ACM Computer Society.

[6] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Computer*, 29(3):210–224, March 2003.

[7] Andrew Eisenberg and Kris De Volder. Dynamic feature traces: Finding features in unfamiliar code. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 337–346, Los Alamitos CA, September 2005. IEEE Computer Society Press.

[8] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[9] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.

[10] Orla Greevy, Michele Lanza, and Christoph Wysseier. Visualizing live software systems in 3D. In *Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization)*, September 2006.

[11] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS '00*, June 2000.

[12] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[13] Dean J. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of ICSE '97*, pages 360–370, 1997.

[14] Jay Kothari, Trip Denton, Spiros Mancoridis, and Ali Shokoufandeh. On computing the canonical features of software systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, October 2006.

[15] Danny Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995)*, pages 342–357, New York NY, 1995. ACM Press.

[16] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[17] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, Cambridge, Mass., 1986.

[18] Alok Mehta and George Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings ACM International Workshop on Principles of Software Evolution*, pages 190–193, New York NY, 2002. ACM Press.

[19] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In Eric Jul, editor, *Proceedings ECOOP '98*, volume 1445 of *LNCS*, Brussels, Belgium, July 1998. Springer-Verlag.

[20] Lukas Renggli. Magritte – meta-described web application development. Master's thesis, University of Bern, June 2006.

[21] Matthias Riebisch. *Towards a More Precise Definition of Feature Models*, pages 64–76. BooksOnDemand Publ. Co. Norderstedt, 2003.

[22] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interacions. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press.

[23] Squeak home page. http://www.squeak.org/.

[24] Squeak IRC client. http://www.preeminent.org/squeak/irc-help/irc-help.html.

[25] Norman Wilde and Michael Scully. Software reconnaisance: Mapping program features to code. *Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[26] Eric Wong, Swapna Gokhale, and Joseph Horgan. Quantifying the closeness between program components and features. *Journal of Systems and Software*, 54(2):87–98, 2000.