Crystallizing Application Configurations

by

Ken (Zanqing) Zhang

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Master of Mathematics

In

Computer Science

Waterloo, Ontario, Canada, 2006

Abstract

Software applications have both static and dynamic dependencies. Static dependencies are those derived from the source code. Dynamic runtime dependencies are established at runtime and may be based on information external to the source code, such as configuration files. Flexible applications commonly rely on configuration to adapt to diverse environments. An application's configuration encodes runtime dependencies between the various parts of the application. Reverse engineering tools have traditionally been based solely on static dependencies extracted from the source code. Neglecting dynamic dependencies encoded in an application's configuration can result in incorrect or incomplete program comprehension. Unfortunately, many applications store their configuration in an ad hoc, unstructured format from which it is not feasible to extract runtime dependencies by traditional reverse engineering. Our work takes advantage of well structured, published configuration formats, such as that of J2EE applications. Using these formats we are able to extend reverse engineering to analyse this previously neglected information. We introduce a technique called **crystallization**, which extracts configuration facts that encode dynamic dependencies. We use these recovered facts to predict and validate dynamic dependencies. Crystallizing configurations has the potential to increase developer productivity by providing better program comprehension.

Acknowledgements

This thesis would not have been possible without the continuous support of my family who gave me the strength and will to succeed.

I would like to thank my supervisor Professor Richard C. Holt for his support and advice. Thanks for listening to and put up with my sometime naïve ideas. I will benefit from his guidance through out my life.

I appreciate the valuable time and encouragement Jingwei Wu provided during my studies.

List of Figures

1 Introduction

1.1 Web Application and Enabling Technologies

The Internet is 15 years old now and the number of users has grown from thousands to millions. Instead of using the Internet for its original purpose such as exchanging files, people are using it to do banking, shopping, etc. Web applications such as e-commerce and Internet banking have become part of our lives. Information that was available at a designated time, at specific locations such as stores and bank branches, is now available anytime, anywhere in the world, given appropriate authentication and authorization. Allowing secure access to information easily has become a vital way to improve quality of service and retain customers. The Internet has become the pervasive vehicle for customer service delivery.

These applications do not exist without the support of enabling technologies such as Java 2 Enterprise Edition (J2EE) and Microsoft .NET Framework. These enabling technologies allow software engineers to develop web applications that interact with enterprise information systems. J2EE and Microsoft .NET technology provide services such as dynamic web page construction, application component life cycle management and configuration based flexible integration. Enhanced with these technologies, the Internet is capable of presenting not only static information such as company addresses, but also dynamic information such as real-time stock pricing that is retrieved from the enterprise information system.

J2EE and Microsoft .NET emerged to support development of reusable, scalable, reliable, and secure enterprise web applications. In addition to supporting development of dynamic web pages, these technologies provide functionality to develop highly decoupled, componentized application modules and allow application integration to be done at a later stage called deployment using configuration. This approach clearly demarcates the responsibility of development and integration. It also allows developers to focus on developing reusable components and deployers to focus on deployment time issues. Application components developed following the framework guidelines can be customized to interact with other components at the integration stage. To simplify our discussion, we will use J2EE as an example of the enabling technologies.

The following is a small example illustrating the difference between the traditional programming model and the J2EE programming model. An application allowing customers to browse products and place orders needs components to support the functionality of "Placing Order", "Processing Order" and "Shipping Order". Figure 1 shows the relationship between these components in the traditional programming model. The *PlaceOrder* component requires a reference to the *ProcessOrder* component. The *ProcessOrder* component requires a reference to the *ShipOrder* component. The *PlaceOrder* component invokes *ProcessOrder* component to process collected order information. Upon successful processing, the order information is further passed to *ShipOrder* component to finalize the order.

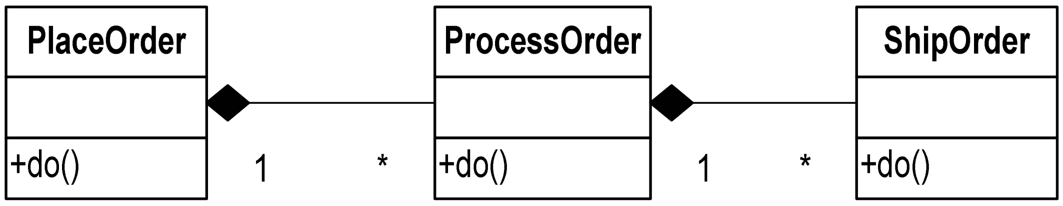


Figure 1: Traditional Component Relationship

With the J2EE framework, the same functionality can be implemented using Java ServerPage (JSP) and Servlet technology. Using JSP/Servlet, all components interact with each other by sending or receiving HTTP [22] requests. For example, *PlaceOrder* collects and submits order information to *ProcessOrder* using HTTP. The processed order information is further submitted to *ShipOrder* to finalize the order.

Each resource such as JSP and Servlet is given a name that is specified by developers. These names are stored in XML configuration files called deployment descriptors. The following is a pseudo snippet from a deployment descriptor showing how *PlaceOrder* is mapped to its implementation class, *PlaceOrderImpl*.

```
…
<component>
  <name>PlaceOrder</name>
  <class>PlaceOrderImpl</class>
</component>
…
```

When an HTTP request is submitted (sent), the receiver of the request is identified using a Uniform Resource Locator (URL) [17]. The URL for JSP and Servlet, as specified by the URL standard, consists of three parts: protocol, hostname and resource name. For web application, the protocol defaults to HTTP. The default value of hostname is the name of the server hosting the application. Both protocol and hostname are optional, the default values will be used when they are absent.

1.1.1 Configuring Web Applications

In a J2EE environment, application components are loosely coupled application components with maximum flexibility and minimum configuration. J2EE components cannot work together to perform an end-to-end business function without additional configuration to help the J2EE runtime resolve required dependencies. Typically, the configuration required is information that guides the J2EE framework to locate dependent components, services and resources.

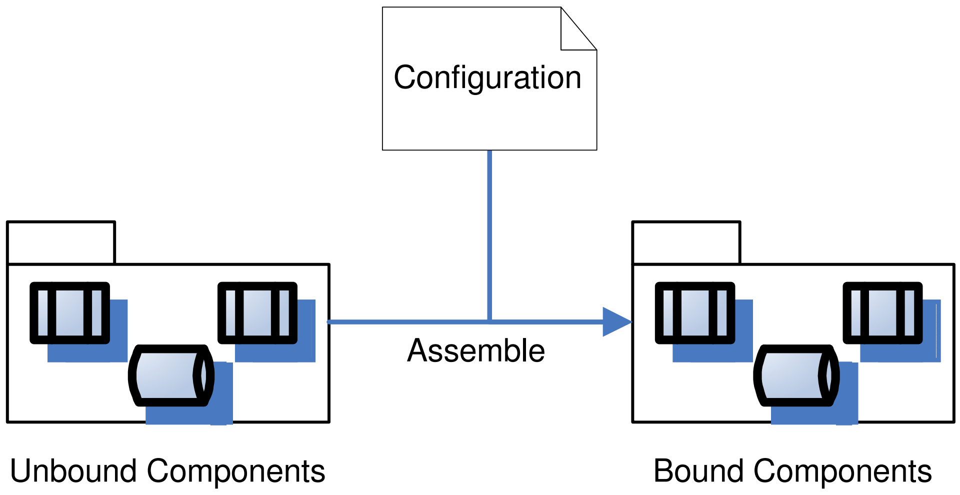


Figure 2: Development Process

Figure 2 sketches the process of assembling J2EE components into applications. In the J2EE programming model, a new role called Application Assembler is introduced. Application Assemblers are responsible for configuring application components they receive from either internal development teams or component vendors. Information such as the name of application components is configured so that the J2EE framework runtime would be able to resolve the required components to perform end to end business functions. All this configuration information is added to the deployment descriptors of the J2EE application. These deployment descriptors are loaded by the J2EE framework when the J2EE application is started.

J2EE Application Assemblers must ensure required resources are available and registered using the right name. For example, as mentioned in previous section, *PlaceOrder* collects order information and submits it to *ProcessOrder* using an HTTP request. Logically, it is obvious that *PlaceOrder* depends on *ProcessOrder*. It is the Application Assemblers' responsibility to ensure the implementation of "*ProcessOrder*" is available and registered with the name "*ProcessOrder*".

1.1.2 Reference by Name

J2EE improves the flexibility of application components by enabling components to reference each other using literal strings that we call *logical names*. J2EE lets programmers define "*logical names*" to which application components can forward control. An application component can forward control to "*Process Order*" component, without knowing the type of the component nor holding an object reference. We call the ability to forward control to application components "***reference by name***". Although it is still possible to reference Java classes using typed object references, reference by a component's logical name is required in order to leverage the flexibility provided by the framework at application integration time. Component logical names are mapped to their concrete implementation when an application is deployed. The component's implementations, normally Java classes, will be looked up and instantiated by the J2EE framework to provide the requested service.

Reference by name, an extra layer of indirection, helps achieve flexibility so that components can be integrated at a later stage based on business requirements; however, it poses problems to application development. Because a component name, e.g., the "*Process Order*", is just a literal string from the compiler's point of view, the compiler can not distinguish "component names" from other regular strings and hence is not able to detect missing components or components missing implementation. The un-typed nature of reference by name makes checking for the existence of a component objects part of the regular development responsibility and adds burden to the developer's already overloaded shoulders. This is not only tedious, but also error prone.

On the other hand, reference by name also has a significant impact on development efficiency and quality. Developers can no longer depend on the mandatory, automatic compiler validation to assure the quality of their programs. They have to first start the application, and then run the business functions that would hit the code to be validated, and finally verify the result to confirm the expected behaviour is achieved. Due to this prolonged validation processes; it takes much longer to verify defect fixes and feature implementations. It is also harder to ensure all expected behaviour is validated due to this manual, tedious, and optional validation procedure.

1.2 Benefits of Crystalization

Reference by name, as one of the most important J2EE techniques to improve flexibility, has also introduced inconvenience to the development society. As compiler aware dependencies being moved into indirect, dynamic, compiler unaware dependencies, getting applications to function correctly is more chanllenging.

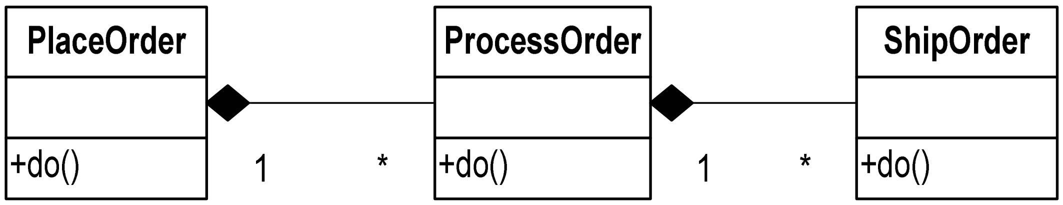


Figure 3: Traditional Component Relationship

For example, developers are very often relying on compilers to type check, among many other validations, the reference from PlaceOrder to ProcessOrder, and the reference from ProcessOrder to ShipOrder. These vadiations must be done maunually by developers when reference by name is used since reference by name is typeless. The crystallization technique understands the reference by name technique and detects the usages of this technique. It further analyzes the detected usages to validate the usages. The crystallization technique not only brings back to the world of "reference by name" the validation capabilities that are available in compilers, but also allows organizations to implement their own validation so that business logic specific validation can be carried out automatically.

1.2 Thesis Contribution

In this research, we developed a reverse engineering method we call ***crystallization*** to explore a new type of information source: application configuration. Crystallization takes into account not only the source code, but also the well structured configuration files that are found in many modern application frameworks. We developed an extensible crystallization framework which can be extended to understand specific application frameworks such as J2EE and Microsoft .NET. The extension framework defines the interface of application framework dependency extractors and validators. The extension framework is capable of detecting installed extractors and validators, and then visualizes extracted and validated dependencies. To illustrate the framework's capabilities we have also implemented a J2EE crystallization extractor and validator. The crystallization method is fully integrated with the Eclipse Integrated Development Environment so that it can provide real-time assistance to programmers without leaving their familiar development environment.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces related work in the area of reverse engineering and program comprehension. We explain J2EE components and services, and the crystallization process for each type of components and services in chapter 3. The implementation of the crystallization framework is illustrated in chapter 4.

We demonstrate crystallization with an example in chapter 5 followed by possible enhancements in the future in chapter 6. Chapter 7 concludes the thesis.

2 Related Work

In this chapter, we introduce research activities related to the topic of this thesis.

2.1 Program Comprehension

Program comprehension has long been recognized as one of the most important activities of software construction. Program comprehension is normally done by reading the documents, source codes, etc. of the program under investigation. *Reading is a key, if not* ***the*** *key technical activity for verifying and validating software work products* [14]. As the life span of applications grows longer and longer, the size of applications is also becoming larger and larger. Many developers with different programming experience and styles may have maintained the application and introduced inconsistencies into the code base. Software developers spend more and more time reading the documentation and source code because of both the inconsistencies and size.

The direct impact of this long and difficult comprehension process is the increased cost of application development and maintenance. The cost of development/maintenance consists of two main parts: first the direct cost such as developer salaries to maintain applications and second, the indirect cost of defects in the application. Any assistance to developers in understanding the application quickly and correctly can directly contribute to lower costs for application maintenance with better quality of the application.

As the scale of these applications become larger, it is common for these applications to be developed by physically disparate teams located on different continents and speaking different languages. It is crucial to be able to quickly and correctly comprehend artifacts, especially foreign artifacts based on information contained in the document and source code. Accurate program comprehension by all developers is a prerequisite to implementation consistency and integrity across the whole application.

2.2 Reverse Engineering

The process of reverse engineering is introduced as an aid in program understanding. This process is concerned with the analysis of existing software systems to make them more understandable for maintenance, re-engineering and evolution purposes. [15] Reverse engineering techniques [6, 10, 11, 12, 13] have long been focusing on extracting facts from source code, documentation, change logs, etc. to reconstruct the "as-built" architecture of the application. The "as-built" architecture gives a high level overview of the structure of the application and the relationship between application components. Code level facts that are not architecturally significant are excluded from the architecture. The architecture is created in the hope to present a consistent, integrated, and easier to understand view of the application under investigation. As shown in Figure 4, the as-built architecture is expected to be one step closer to the mental model to be constructed by software developers during the program comprehension process.

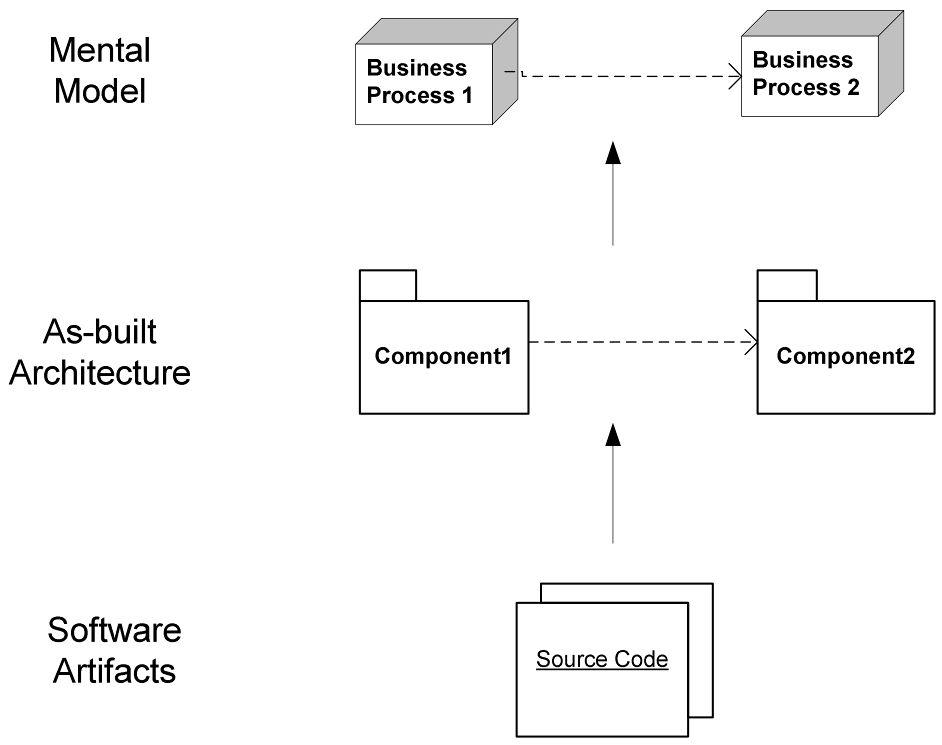


Figure 4: As-built Architecture and Program Comprehension

Some of the most often extracted information includes function calls, object inheritance, and directory structures. The extracted facts such as function calls between components represent the relationships between application artifacts and the number of the function calls suggests how tightly the two components are coupled. The facts are further processed using various clustering techniques to collect artifacts into logical groups.

2.3 Software Visualization

Among the methods to support software development that have been proposed in literature, software visualization has long been considered as a technique to aid comprehension. Much effort has been placed in this area. Tools such as LSEditor[18] and Rigi[19]. have been developed to assist developers in investigating the static relationships between application modules interactively. These tools visualize software by relying on static facts extracted from the source code based on syntax. Some of these fact extractors are constructed by modifying the compiler so that instead of generating machine code, it generates facts in a format can be used by the tools. LSEditor supports multiple different algorithms to cluster application components. Users are allowed to try different layout and clustering methods so that application components are visualized in such a way that aligns with the mental model they constructed during the program comprehension process. The cluster of the application components and the mental model are dynamic which could change over time based on the information and the better understanding of the source code.

There are also tools such as GROOVE [26] that analyzes and visualizes the runtime aspects of applications. GROOVE captures runtime events such as *Instance Create* and *Method Invoke* to construct a presentation of the runtime attributes of an application. GROOVE is a very useful tool for tracing and debugging applications. Another type of tools that are used to trace runtime attributes is UML sequence diagram generators. These tools capture method invocations and use this information to generate UML sequence diagram. Althought it is useful, sometimes the generated UML diagram can be very difficult to follow due to the amount of detail information, for example, all the calls to library methods that are not of interest.

2.4 State of the Art

Much effort has been spent to improve the integreated development environment for J2EE based applications. Many tools incluing Eclipse, Rational Application Developer[28], NetBeans[29], OptimalJ[27], IDEA[30], just to name a few, provides great help to developer to improve productivity and quality.

One of the most important functionality that helps developer quickly comprehend the source code is the ability to quickly navitage the source code. The faster that developers can nagivate the source code, the quicker they would be able to find the information needed to comprehend the source code. IDEs have gone a long way in this direction to assist program comprehension. For example, when a Java class is opened in a Java source code editor in Eclipse, finding out the definition of a Java class used in the code is only a single mouse click away. Besides class definition, it is also extremely easy to look for all the references to a class, all of the invocations to a method, all the subclasses, etc. The overall view of code, for example, all the invocation to a method, gives developers information how the method is used. It not only helps developers understand the purpose of the method, but also allows them to evaluate the impact of change if the method is to be changed.

Many IDEs, including those mentioned above, cooperate color highlighting, integrated compilation, and unified error report to improve development productivity and quality. The Web Tool Eclipse Project goes one step further to validate configurations Java classes that are used in JSPs. This is done by extracting Java snippets embedded in JSPs and statically analysing it. Althought it is able to help developers validate the format of certain types of configuration files, it is not able to analyze and integrate the information from multiple configuration files to validate the correctness of the application.

3 Dependency Crystallization

The J2EE specification specifies a set of standard configuration files that are used to integrate J2EE applications. Application components that are decoupled at compile time may interact due to configuration. The way that J2EE framework achieves this flexibility is by referencing J2EE components using their logical names and by allowing application deployers to associate logical names with J2EE components when applications are configured and deployed.

To help developers understand applications, the reverse engineering community commonly uses analysis techniques [6, 10, 11, 12, 13] to extract dependencies from the application's source code and then uses these dependencies to help developers understand the relationships between its components. Unfortunately, the static dependencies derived from an application's source code may be insufficient to reveal key relationships between its components. This is due to external information such as its configuration adding or modifying relationships between the components. The information encoded in an application's configuration can be essential to the comprehension of a program. But, in many cases, the ad hoc, unstructured format of this configuration information makes it difficult to understand. This is especially difficult in that each application could store its configuration in its own particular manner. Integrated Development Environments (IDEs) typically do not understand an application's configuration information and are thus unable to help developers ensure that the configuration is correct. Fortunately, application frameworks such as J2EE have a well structured, published format to store configuration information. This makes developing program comprehension tools, including IDEs that leverage configuration information, possible.

Our technique, which we call *crystallization*, enhances *static* dependencies from source code, with *dynamic* dependencies [10, 11, 12. 13] encoded in the configuration. This more complete extracted information allows us to deduce and validate dynamic dependencies. While our discussion and implementation is based on J2EE, our technique can be applied to other frameworks which use structured formats to configure runtime application dependencies.

We will use an example to illustrate one of the ways that the configuration of a J2EE application determines dynamic dependencies. In Object Oriented (OO) software, a reference to an object instance is used to invoke methods of the object. J2EE generalizes this approach in that the name of a component is used to invoke predefined methods in particular components. As illustrated by Figure 5, in traditional OO programming class *Foo* references object *BarImpl* which implements the *Bar* interface. The ***new*** construct creates the object instance. The reference to the new object instance is stored in *ref*, which is used to invoke a method, for example, doPost().

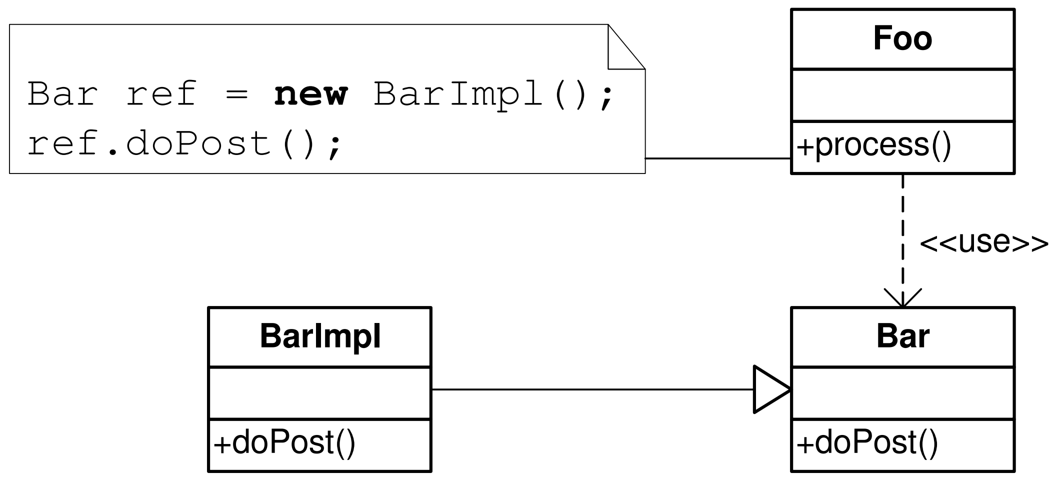


Figure 5: Static Dependency in Traditional OO

J2EE stores configuration information in deployment descriptors (DDs), which are XML files. Deployment descriptors contain definitions of J2EE components including their names, implementing Java classes and other runtime attributes. Each component is defined in a component type tag containing a name and an implementing class element. As illustrated in the pseudo snippet below, a component named *BAR* with implementation class *BarImpl* is declared.

```
…
<component>
  <name>BAR</name>
  <class>BarImpl</class>
</component>
…
```

In J2EE, it is encouraged to *reference* components *by name* instead of by object reference to acquire the services they provide. The use of component name allows application integrators and deployers to easily change the binding of the name. In other words, application integrators and deployers can change the implementation associated with a logical name dynamically. The use of the name of a component implies a dynamic dependency on that component, and hence on the implementing class of the component. For example, at runtime the following statement

```
HttpServletResponse.sendRedirect(“BAR”)
```

which references component *BAR* by name, triggers J2EE to create an instance of *BarImpl*, the implementing class of component *BAR* and to invoke *BarImpl*'s predefined method, *doPost()*. If the configuration is changed so that the implementing class of *BAR* becomes *BarImpl2*, an instance of *BarImpl2* would be created and its *doPost()* would be invoked. This is an example of the flexibility J2EE provides to switch implementations without recompilation. In this example, there is neither an object reference nor a function call involved and thus static analysis techniques would be unable to capture the dependency from component *BAR* to the implementing class *BarImpl*.

These dynamic configuration dependencies are neither captured nor indicated by tools such as compilers and IDEs when there is a mis-configuration. This increases the likelihood that new members of a development team who are not familiar with the code base will make mistakes. For example, the newcomer, in the process of refactoring the source code, may change the name of a class without knowing the dependencies on the name of the class. This would break dynamic dependencies without introducing any compilation errors. Components that are still using the old name to acquire services from the component would fail since the J2EE framework would not be able to resolve the name. One common approach is to manually inspect the code base to catch mis-configuration, which is time consuming and tedious.

In order to tackle the difficulties brought in by dynamic dependencies, tools need to understand not only the language used to construct the components, but also the configuration that establishes runtime dependencies so that mis-configured dependencies will be captured at an earlier stage.

Our crystallization processes first extracts component definitions from deployment descriptors and component name references from the source code, and then resolves these component name references and component definitions in a way similar to how the J2EE framework runtime resolves dynamic dependencies. The crystallization process notifies developers to correct any erroneous dependencies such as references to components that do not exist.

3.1 J2EE Configuration

Our experimental work is based on the J2EE framework. We will explain J2EE configuration and how it introduces dynamic dependencies between components. This is not meant as a J2EE tutorial but as an introduction to components that are difficult to understand or maintain due to dynamic configuration dependencies.

J2EE provides an architecture framework for enterprises to build multi-tier distributed applications. As shown in Figure 6, J2EE provides JavaServer Page (JSP) [7, 9] and Servlet technology to implement web tier components. It also provides Enterprise JavaBean (EJB) [8, 9] technology to implement business tier components. J2EE supports component communication and interaction using Java Messaging Service (JMS) technology, which supports application modularity, scalability and flexibility.

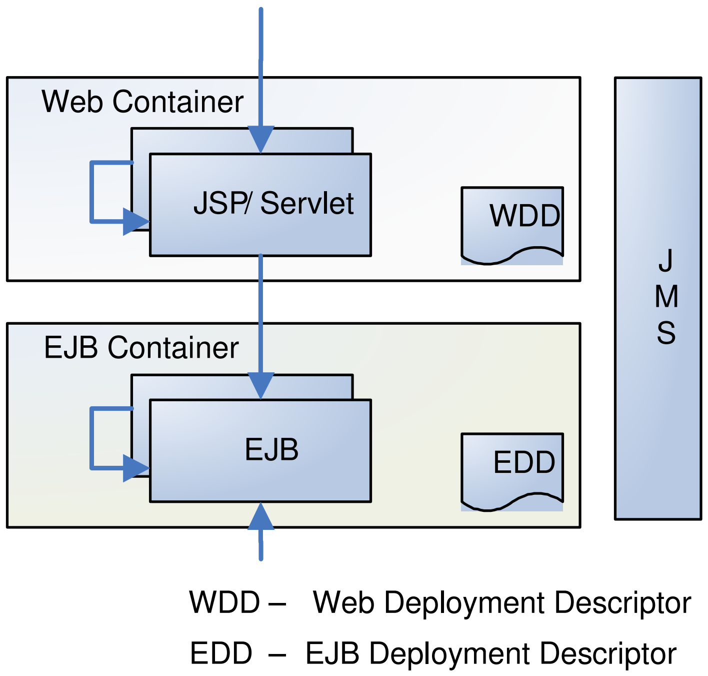

Web Container
JSP/ Servlet
WDD
EJB Container
EJB
EDD
JMS
WDD – Web Deployment Descriptor
EDD – EJB Deployment Descriptor

Figure 6: J2EE Components and Services

3.1.1 Web Tier Components and Configuration

JavaServer Page (JSP) and Servlet are technologies used to implement web tier components of J2EE applications. These components run in a Web Container as shown in Figure 6. JSP and Servlet technology simplify web based user interface development. JSPs are comprised of HTML intermingled with scriptlets of Java code which dynamically generate HTML code. The static and dynamically generated HTML code works together to present user interface in browsers. Servlets are written in pure Java. While they can be used for the same purpose as a JSP, e.g. dynamically generating HTML code without having any static HTML code, their intended purpose is to provide business workflow control. This allows JSP developers to take HTML pages that are designed by graphic designers and add business logic such as retrieving data dynamically without dramatically altering the HTML code and without worrying about the page layout. User input collected from HTML forms [16], either created statically or dynamically by JSPs' Java scriptlet, is normally submitted to Servlets. These Servlets collate the input and invoke business logic components such as EJBs to process the input and redirect to a JSP to present the result of the process.

```
<servlet>
    <servlet-name>BARServlet</servlet-name>
    <servlet-class>example.web.Bar</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>BARServlet</servlet-name>
   <url-pattern>BAR.DO</url-pattern>
</servlet-mapping>
```

Figure 7: Snippet from Web Deployment Descriptor

Each JSP or Servlet is assigned a name in the web deployment descriptor (WDD). As illustrated in Figure 7, servlets are defined in a servlet tag, <*servlet*>, which contains a servlet name element, <*servlet-name*>, and a servlet class element, <*servlet-class*>. The servlet name is an internal name, e.g. *BARServlet*, which is used in the <*servlet-mapping*> element to define the external name that the servlet will be referenced by. The servlet class defines the implementing class. A servlet definition is followed by a servlet mapping tag, <*servlet-mapping*>, containing a mapping from the internal name to the external name, e.g., from *BARServlet* to *BAR.DO*. A component's name, optionally prefixed with the name of the server hosting the J2EE application forms a Unified Resource Locator (URL)[17], which is used to reference the component. The server name prefix is only required only when referencing component in a different context, for example, another component hosted on a different server.

As elaborated, a reference, possibly in URL format, to a web component name defined in the web deployment descriptor implies a dynamic dependency on the implementing class of the component. From the traditional static analysis' point of view, the references to components' names are just regular string literal references.

3.1.2 Business Tier Components and Configuration

EJBs [8] are business tier components and run in EJB Containers as depicted in Figure 6 previously. EJBs can be deployed on multiple servers and the J2EE framework provides load balancing and fail-over protection to improve service performance, availability and scalability. In addition to these services, J2EE also provides transaction management facilities to applications.

Commonly, large-scale applications deploy EJBs on multiple servers. These servers work together as a cluster waiting for service requests. A designated server running the J2EE framework dispatches service requests using algorithms such as round robin to all the servers in the cluster. Since all services are available at more than one server, there is no single point failure in the application. Increased scalability of the application can also be achieved by adding more servers to the cluster and deploying EJBs.

At runtime, EJB service requesters ask the J2EE framework for an EJB instance by *name*. Figure 8 shows a snippet from an EJB deployment descriptor that defines an EJB named *BAREJB* as specified in the <*ejb-name*> tag. The *BAREJB* provides services defined in the remote interface, *example.ejb.BarRemote* as specified in the <*remote*> tag. The services *BAREJB* provides are implemented in *example.ejb.BarImpl* as specified in the <*ejb-class*> tag.

```
<enterprise-beans>
<session>
          <display-name>BAREJB</display-name>
          <ejb-name>BAREJB</ejb-name>
          <remote>example.ejb.BarRemote</remote>
          <ejb-class>example.ejb.BarImpl</ejb-class>
<session-type>Stateful</session-type>
```

Figure 8: Snippet from EJB Deployment Descriptor

Each EJB service requester has an object reference to the EJB remote interface, *example.ejb.BarRemote*, and the implementation logically "implements" the remote interface. By logically implementing the remote interface, we mean the implementing Java class does not have to inherit the remote interface using the Java keyword "**implements**" although it does contain implementation of all the methods defined in *example.ejb.BarRemote*.

J2EE discourages EJB implementing classes from implementing EJBs by inheriting the remote interface. Instead, it is the configuration, namely the EJB Deployment Descriptor, which glues the parts of EJBs together. The references to remote interface are bound to object instances of the implementing class at runtime. The J2EE framework picks the instance to be bound to remote interface references from a pool of instances of implementing classes that are initialized when the J2EE framework is started. All method invocations are dispatched to the implementing class. Since the implementing class does not implement the remote interface, an instance of the implementing class "*is*" not an instance of the remote interface and the invocation dispatch is not performed in the same way as *polymorphism* where implementing classes implement the remote interface.

The components holding a reference to the remote interface require the implementing classes to present at runtime in order to perform its function. Hence, a reference to the remote interface implies a dynamic dependency on the EJB implementing class. Because the implementation implements the remote interface "logically" and hence does not have any syntactic relationship to the referencing component, static analysis techniques are unable to capture these dependencies.

3.1.3 Java Messaging Service (JMS)

JMS allows J2EE components to communicate by exchanging synchronous or asynchronous messages using message queues or topics. JMS queues may have multiple senders and multiple receivers. The messages sent to a JMS queue are guaranteed to be delivered to a receiver once and only once. JMS Topic is a subscription based messaging model. It delivers all messages sent to a topic to all of its subscribers to the topic. While we focus on JMS queue in our research, a similar approach can be applied to JMS topic based communication.

This message based communication model decouples message senders from message receivers. At compile time, each message sender knows the name of queue to which it is sending and each receiver knows the name of queue from which it is receiving. However, a sender does not in general know which receiver will receive a given message, nor does a receiver know which sender sent a message. This allows different developers, possibly different vendors to work on senders and receivers separately although a common message format must agreed upon. The ability to ensure the persistence of messages allows the message sender and receiver to run asynchronously. These persistent messages are delivered when receivers become available.

Another benefit of using JMS services is the ability to achieve scalability. As you might have noticed that JMS queues allow multiple message receivers, it is a common practice to increase the number of receivers, possibly running on different servers, to increase the throughput of message processing. Adding more receivers can increase the availability of the application since no single receiver failure would bring down the application.

As illustrated in Figure 9, *Foo* and *Bar* are not statically dependent on each other. At runtime, *Foo* and *Bar* ask the J2EE framework for a reference to a common queue by invoking a predefined method. This queue object is used to send or receive messages. It is clear that when *Foo*, the sender and *Bar*, the receiver are referencing the same queue, *Foo* has a dynamic dependency on *Bar*.

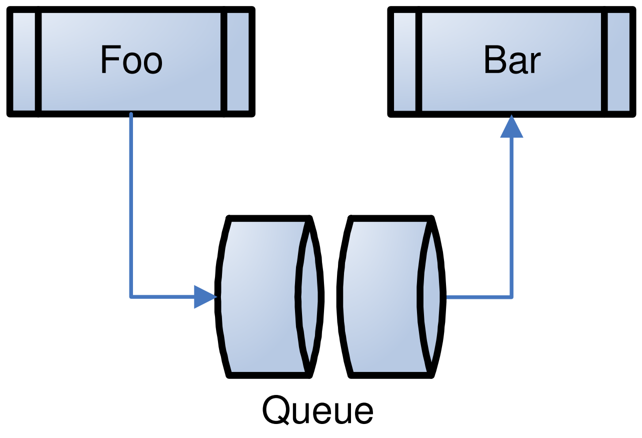


Figure 9: JMS Communication

Although there are no configuration files required for components using JMS, we consider the name of the JMS queue to be the configuration information since this can be changed without affecting expected behaviour of the component. In fact, the name of the JMS queue used by components is normally stored in a configuration file as key value pairs such as *Order.Queue.Name=orderQueue*, although this is not specified by J2EE specification as a standard configuration convention.

3.2 J2EE Application Development Challenges

J2EE provides great flexibility to configure application behaviour to meet changing business requirements without recompilation; it also introduces additional complexity to application development. Because components are statically decoupled and no longer have syntactic dependencies that are visible to regular IDEs, dynamic configuration dependencies that are established at runtime are often not apparent to developers.

For example, large, evolving projects with web interfaces may have many, possibly thousands of ever changing JSPs. Due to changing requirements, JSPs can be obsolete very quickly. Very often, these JSPs are not removed from the code base immediately, which results in many unused JSPs. Without assistance from tools that crystallize dynamic configuration dependencies, it is difficult to locate and remove these unused JSPs due to these unapparent dynamic dependencies.

EJBs also pose problems, since the implementing class is not required to inherit the remote interface. Missing implementations of exposed remote interface would not cause compilation errors. Mis-configuration such as a typo in the name of the implementing class would not be detected immediately either. Errors may not surface until the application is up and running. This delays detection of these errors and consequently lowers programming productivity.

Although JMS technology is not difficult to understand conceptually, the way components interact using JMS is not recognized by traditional IDEs. Developers have to inspect the source code manually in order to determine the communication channel. Without a tool that understands JMS communication and configuration, JMS communication related problem determination is time-consuming and error prone.

3.3 Crystallizing Configuration Dependencies

Understanding dynamic configuration dependencies is a challenge facing J2EE application developers. Existing Java compilers and IDEs do not warn developers of erroneous dependencies resulting from mis-configuration. To assist developers in overcoming these challenges we developed a process to crystallize the configuration information into an understandable form. The crystallized configuration information is further processed and graphically presented to users so that errors in the artifacts they are working on are captured easily.

We accomplish this using our crystallization process as follows. First, we analyze the J2EE technology and its configuration to identify configuration and coding patterns that could result in runtime dependencies. Second, we search for the identified patterns in the source code and configuration to predict dynamic dependencies. Finally, recovered dependencies are graphically presented to developers in an easily consumed form using approaches such as color highlighting.

3.4 Crystallization Process

In order to crystallize dynamic configuration information, we need to understand *what* and *how* J2EE components are invoked at runtime. Different types of J2EE components are invoked in different ways. J2EE provides APIs to invoke J2EE components. The name of the invoked J2EE component is normally passed to the API methods as parameters. The name is resolved into either Java classes such as Servlet or EJB implementing class or J2EE services such as JMS queue, based on the type of the invocation. J2EE configuration files, e.g. Deployment Descriptors, are the central place for information used for the resolution.

Figure 10 shows the type of documents that are included in our process. Java Server and HTML Pages, Web deployment descriptors, and EJB Deployment descriptors are analyzed using our crystallization parsers. The Java source code is analyzed using traditional reverse engineering methods to extract static dependencies. Furthermore, our crystallization parser is also applied to Java source code to capture parameters that are used to invoke special APIs. The information from these parameters is later used in the crystallization process to determine the target of the invocation.

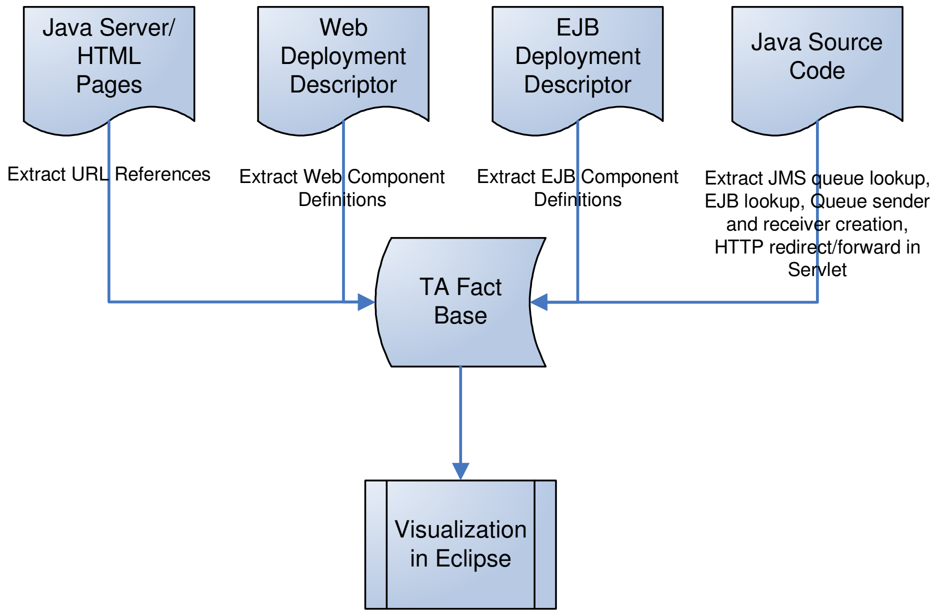

Java Server/ HTML Pages
Web Deployment Descriptor
EJB Deployment Descriptor
Java Source Code
Extract URL References
Extract Web Component Definitions
Extract EJB Component Definitions
Extract JMS queue lookup, EJB lookup, Queue sender and receiver creation, HTTP redirect/forward in Servlet
TA Fact Base
Visualization in Eclipse

Figure 10: Crystallization Process

3.5 Dependency Notation and Operation

There are several choices including Graph eXchange Language (GXL) and Tuple Attribute[2] (TA) in terms of the format of record of extracted facts. The GXL format is XML based with hierarchical information. Although GXL provides better readability, it is designed for data exchange between tools, across various platforms. It is difficult and inefficient to apply logics on fact represented in GXL format to extract high level information, especially when the number of the facts is large. The TA format is not only easy to understand and simple to represent, but also feature rich. Since each TA record represents a relation, all relational algebra operations can be applied to TA records to extract high level relationship. Grok[23], a efficient and easy to use relational calculator that applies relational algebra operations to TA records, is widely used in the reverse engineering society to manipulate facts recorded in TA format to calculate architectural level relations. In our implementation the extracted dependencies such as method invocations are stored in TA format as follows:

```
relation-type origin destination
```

The left part in the tuple is the type of the relation, followed by the origin and the destination of the relation. For example, a reference to URL *bar.do* in *foo.java* is recorded as

```
url-reference foo.java bar.do
```

This relation can be read as: there is a `url-reference` from the `origin` *foo.java* to the `destination` *bar.do*.

A web component named *bar.do* with implementing class, *example.web.Bar* is stored as

```
    web-definition bar.do example.web.Bar.
```

This relation can be read as: there is a web component definition for *bar.do* which is implemented by the class *example.web.Bar*.

In order to find out the real target of the `url-reference` from *foo.java* to *bar.do*, we need to find out the definition of *bar.do* which is available in the `web-definition` relation. The composition of `url-reference` and `web-definition` reveals the real target of the `url-reference`, namely the runtime dependency between *foo.java* and *example.web.Bar*. The following formula illustrates the steps to recover the real target mathematically. The composition of relations, e.g. `url-reference` and `web-definition`, is noted as `url-reference ∘ web-definition`.

```
url-reference origin-comp dest-component
web-definition web-component implementation
url-reference ∘ web-definition origin-comp implementation
```

The employment of TA allows us to apply relational calculus operations such as union, subset, and composition [2] on the set of extracted references to obtain higher level, meaningful relations.

3.6 TA Schema for configuration dependencies

The following is a list of the relations that is created by the crystallization framework, in the form of TA schema. Each relation is followed by a brief description.

```
SCHEME              TUPLE

url-reference       file            component
//web component is referenced in file using url

web-definition      file            component
//web component is defined in file

queue-send          file            queue
//file sends message to the queue

queue-recv          file            queue
//file receives message from the queue

ejb-implementation  file            component
//EJB component is implemented by file

ejb-remote          file            component
//EJB component remote interface is defined in file

ejb-reference       file            component
//EJB component is reference by file
```

The crystallization framework keeps extra information such as line number of relations. Although this kind of information can also been expressed in TA record, we decided to make it an attribute to each relation. The line number information allows us to accurately pinpoint invalid relations in the visualization component of crystallization framework.

3.7 Important J2EE APIs

As a complex framework, J2EE is not different from any others that introduce yet another set of APIs for interactions between both application components and J2EE services. Method invocations to particular J2EE APIs, such as Servlet request dispatches and JMS queue sender and receiver creation, contain semantic information that are useful for dependency crystallization. The semantics of the invocation and the parameters to the invocation, together with information gathered from deployment descriptors, enable us to locate target J2EE components to be invoked at runtime and hence allow us to predict possible dynamic dependencies in the source code.

Servlets are designed for business workflow control. Servlets decide the next step to be carried out based on the context and the result of the current process. It is an essential requirement to be able to forward or redirect requests to another servlet for further processing. J2EE introduced a class called *RequestDispatcher* to perform this action. In order to forward or redirect requests, the Servlet creates a *RequestDispatcher* object and calls its *forward()* method. The URL of the target web component is passed to the invocation as parameters. These invocations indicate dependencies from the Servlet to the web component identified by the URL.

The way some APIs are invoked can be used to distinguish the type of the component. For example, JMS senders and receivers can be distinguished by the API method invoked. Senders and receivers are created by invoking *javax.jms.QueueSession.createSender()* and *javax.jms.QueueSession.createReceiver()* respectively. Based on this difference, we are able to determine the direction of the communication and hence the direction of the dependency.

Table 1 summarizes the types of reference that exist in different types of components. For example, we can find `component-url` type references in web components defined in HTML and JSP pages. On the other hand, if we find a invocation to `forward()` method in a JSP or a Servlet, we know the JSP or Servlet is referencing a web component and the name of the component is the parameter that is used in the method invocation.

Component Type	**Types of References**	**Sources**
Web Component	"component-url"	HTML, JSP

	sendRedirect()
	forward()
		Servlet

EJB	ejb.RemoteInterface	Java

JMS	Queue queue = …
	createSender();
	createReceiver();
		Java

Table 1: Component References and Sources

3.8 Crystallizing Web Dependencies

The goal of crystallizing web dependencies is to recover the relationship between HTML, JSP and Servlets. Since all web components are referenced using their URL, the first step is to extract URLs contained in the web components. because the extracted URLs are not physical artifact of an application, they need to be further resolved into physical files to reveal the real relationship.

We need to identify the places that URL might be used in order to extract URLs. There are various places that URL may be used:

- HTML links, e.g., <a href = "**bar.do**">bar</a>
- HTML form action targets <form action="**bar.do**">
- JSP forward tags, a special tag used by JSP to forward HTTP requests
- Servlet request redirects and forwards, invocations to *sendRedirect()* and *forward()* introduced in section 3.7

Extracting the URL references in HTML, JSP and Servlet is the first step to crystallizing web dependencies. For this step, We have built three parsers. The HTML/JSP parser extracts references to URLs. The Servlet parser captures method invocations and the parameters to *sendRedirect()* and *forward()*. The deployment descriptor parser extracts web component definitions.

The following source code snippet from a Servlet, *foo.java*, (can also see this in a JSP) shows how a HTTP request is dispatched to "*bar.do*" by invoking the *RequestDispatcher.forward()* method. First, a `ServletContext`, `context`, is obtained. This object contains the information of the context where the Servlet is running. Second, a `RequestDispatcher`, `dispatcher`, is created by invoking `getRequestDispatcher("bar.do")`. The `dispatcher` object contains the URL of target, e.g., `bar.do`. The forward is finally done by calling the `forward()` method to pass all the information that is originally passed to *foo.java*. URL reference is done by capturing the parameter that is used in the method invocation `getRequestDispatcher("bar.do")`.

```
ServletContext context =  getServletConfig().getServletContext();
RequestDispatcher dispatcher = context.getRequestDispatcher("bar.do");
dispatcher.forward(req, resp);
```

With the URL extracted from previous step, we still do not know which physical artifacts, e.g., which HTML, JSP or Servlet that is referenced. In other words, what exactly is **`bar.do`**. We cannot answer this question without looking in to the web deployment descriptors. In the deployment descriptor shown in Figure 11, "*bar.do*" is implemented by *example.web.Bar*. An HTTP request to "*bar.do*" results in the invocation of the predefined method in *example.web.Bar*, the *doPost()*.

```
<servlet>
    <servlet-name>BARServlet</servlet-name>
    <servlet-class>example.web.Bar</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>BARServlet</servlet-name>
   <url-pattern>BAR.DO</url-pattern>
</servlet-mapping>
```

Figure 11: Snippet from Web Deployment Descriptor

The extracted URL reference in TA notation, `url-reference` *foo.java bar.do*, is composed with the web component definition, `web-definition` *bar.do example.web.Bar*, which reveals a dynamic dependency from *foo.java* to *example.web.Bar* as shown in Figure 12.

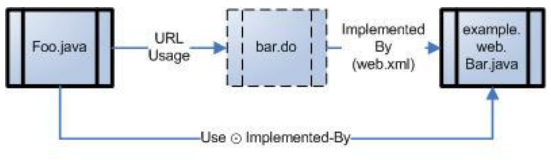


Figure 12: Crystallized Web Dependency

3.9 Crystallizing EJB Dependencies

EJB callers possess references to the remote interface defined in the EJB deployment descriptor. The remote interface defines the business methods that are exposed by the EJB.

As shown in the following code snippet, the EJB caller has a reference *bean* to the BarRemote interface as depicted in Figure 8 previously. The reference is bound to an instance of the EJB implementation class by calling the method *getInstance()*. We neglect the details of binding the remote interface reference to an instance of the implementing class since it is not related to how dependencies are crystallized. The information that is important to our crystallization process is the presence of the remote interface reference and the usage of this reference, e.g., the declaration of the remote interface variable as shown below and method invocations using the reference.

```
BarRemote bean = getInstance();

bean.method1();

bean.method2();
```

The EJB parser captures references to remote interfaces of EJBs. As shown in Figure 13, the remote interface reference relation is composed with EJB definitions found in EJB deployment descriptors to reveal the real dependency between the referencing component and the EJB implementing class.

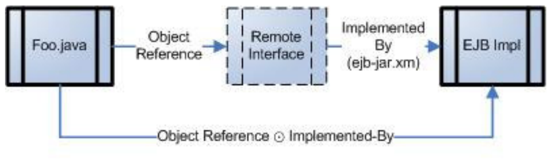

Foo.java
Object Reference
Remote Interface
Implemented By (ejb-jar.xm)
EJB Impl
Object Reference ⊙ Implemented-By

Figure 13: EJB Interaction

3.10 Crystallizing JMS Dependencies

In order to communicate through a JMS queue, senders and receivers must have a reference to the JMS queue. Capturing JMS interactions starts with capturing JMS queue references. JMS queue references reveal all components; consisting of senders and receivers. However, we are not only interested in the participants of communication but also their relationships. JMS participants invoke the *createSender()* and *createReceiver()* methods, capturing these invocations allows us to separate them into senders and receivers.

The following code snippet shows how *QueueSender*, the sender, and *QueueReceiver*, the receiver, are created. JMS participants ask J2EE for a reference to a queue instance. To initiate communication a connection must be established followed by opening a session.

```
Queue queue = (Queue)  context.lookup("OrderQueue");
QueueConnection conn = createConnection();
QueueSession session = createSession(conn);
QueueSender qSender = session.createSender(queue);
QueueReceiver qReceiver = session.createReceiver(queue);
```

JMS queue communication involves sender(s) and receiver(s); we need to determine which sender(s) is associated with which receiver(s). This is achieved by matching sender(s) and receiver(s) that communicate through the same queue. Since a reference to a JMS queue is obtained from J2EE by queue name as follows:

```
Queue queue = (Queue) context.lookup("OrderQueue");
```

In TA notation, the above fact is recorded as:

```
queue-send Foo.java OrderQueue
queue-recv Bar OrderQueue
```

The composition of these relations yields the following dependency:

```
Jms-dependency foo.java bar.java
```

This is illustrated in Figure 14.

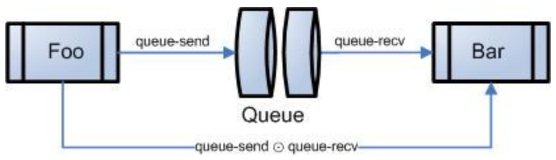


Figure 14: Crystallized JMS Interaction

4 Crystallization Implementation

In this chapter we discuss the implementation of crystallization process. The crystallization tool is implemented as plugins to the popular Integrated Development Environment, Eclipse. The implemented crystallization tool also exposes extension points so that it can be extended to extract and validate other configurations.

4.1 Requirements of Crystallization Tooling

The crystallization tool was created to improve software development productivity. It should not only help improve development efficiency, but also development quality. Building a defective system that requires massive maintenance efforts post-development is more expensive than developing it correctly on the first try. Based on the requirements of Computer-Aided Software Engineering (CASE) tool outlined by [20], we collected the requirements of the crystallization tool:

1. An interactive presentation providing a consistent, "friendly" user-interface to analysis of the inter relationship between application artifacts;
2. Automatic validation of changed artifacts without manual user intervention with automatic notification whenever invalid relationships are found;
3. A navigation system that allows the user to traverse the relationship graph easily;
4. A graphical presentation of the relationship of the "context", e.g. the artifacts that the user is currently working on.

4.2 Eclipse Introduction

We choose Eclipse as the foundation of our implementation for numerous reasons. Eclipse is an extensible, open source platform for development of highly integrated tools. The Eclipse platform, when combined with Java Development Tools (JDT), offers many of the features you would expect from a commercial-quality IDE: a syntax-highlighting editor, incremental code compilation, a thread-aware source-level debugger, a class navigator, a file/project manager, and interfaces to standard source control systems. Such as CVS and ClearCase.

Despite the large number of standard features, Eclipse is different from traditional IDEs in a number of fundamental ways. The most interesting feature of Eclipse is that it is designed to be platform and language neutral. In addition to the eclectic mix of languages supported by the Eclipse consortium such as Java, C/C++, and Cobol, there are also projects underway to add support for languages as diverse as Python, Eiffel, PHP, and C# to Eclipse. Figure 15 illustrates the architecture of the Eclipse platform.

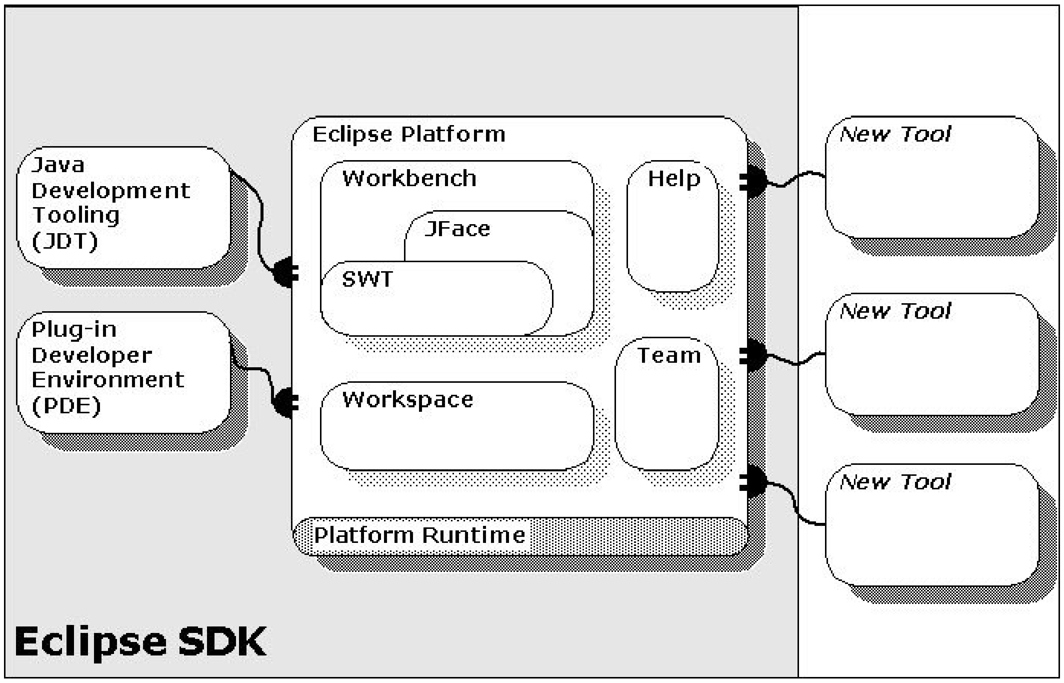


Figure 15: Eclipse Architecture [1]

A plug-in is a structured component that contributes code (or documentation or both) to the platform and describes it in a structured way. Plug-ins can define **extension points**, well-defined function points that can be extended by other plug-ins. Using a common extension model provides a structured way for plug-ins to describe the ways they can be extended, and for client plug-ins to describe the extensions they supply. Defining an extension point is much like defining any other API. The key difference is that the extension point is declared using XML instead of a code signature. Likewise, a client plug-in uses XML to describe its specific extension to the system.

4.3 JDT Introduction

The JDT is implemented as a group of plug-ins[1]. JDT adds Java specific behaviour to the Eclipse framework and contributes to the Eclipse UI Java specific views, editors and actions. JDT is divided into UI plug-ins and Non-UI core plug-ins. Besides implementing features supporting Java development, both the UI plug-ins and Non-UI plug-ins expose extension points that allow third parties to extend JDT. Figure 16 is a diagram illustrating the relationship between JDT and Eclipse with a list of extension point that are implemented by JDT.

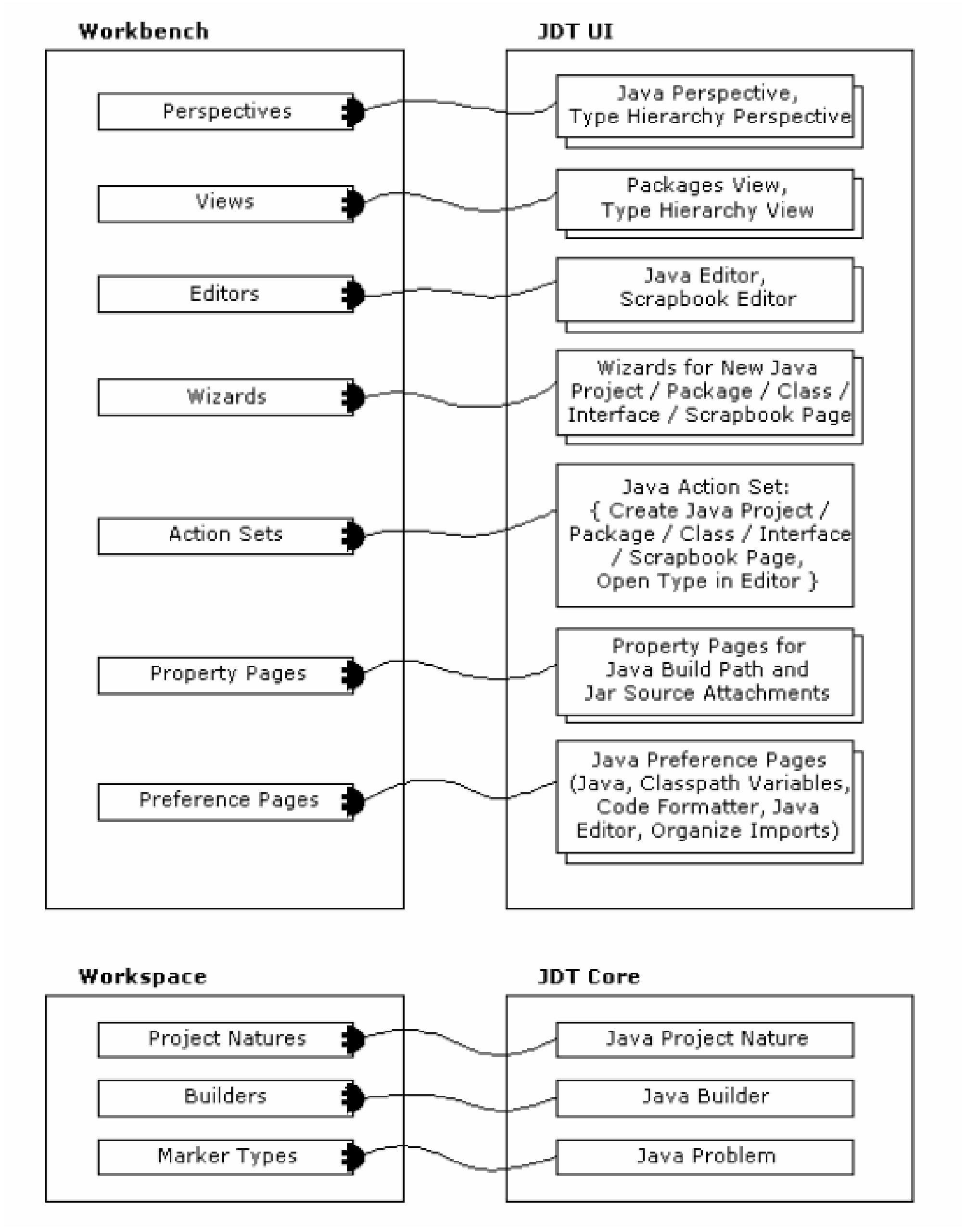

Workbench
JDT UI
Perspectives
Java Perspective, Type Hierarchy Perspective
Views
Packages View, Type Hierarchy View
Editors
Java Editor, Scrapbook Editor
Wizards
Wizards for New Java Project / Package / Class / Interface / Scrapbook Page
Action Sets
Java Action Set: { Create Java Project / Package / Class / Interface / Scrapbook Page, Open Type in Editor }
Property Pages
Property Pages for Java Build Path and Jar Source Attachments
Preference Pages
Java Preference Pages (Java, Classpath Variables, Code Formatter, Java Editor, Organize Imports)
Workspace
JDT Core
Project Natures
Java Project Nature
Builders
Java Builder
Marker Types
Java Problem

Figure 16: Key Connections between JDT and Eclipse [1]

4.4 Crystallization Framework

The crystallization framework is implemented as an extension to the JDT and is seamlessly integrated with the Java views, editors and actions. Our tool does not change the pattern of typical programming activities in JDT.

The parsers we have implemented extract dependencies from HTML files, JSPs, Java sources and deployment descriptors. These parsers are implemented as extensions to `org.eclipse.core.resources.builders` extension point. We integrated these parsers into the JDT, and extended the JDT Java editor with the ability to traverse not only static relationships that are visible to the compiler, but also relationships that are crystallized from the configuration. The integration of these parsers within an IDE ensures the instant accessibility of the crystallization technique without switching over to a separate tool. Figure 17 shows the workflow of crystallization process in our implementation.

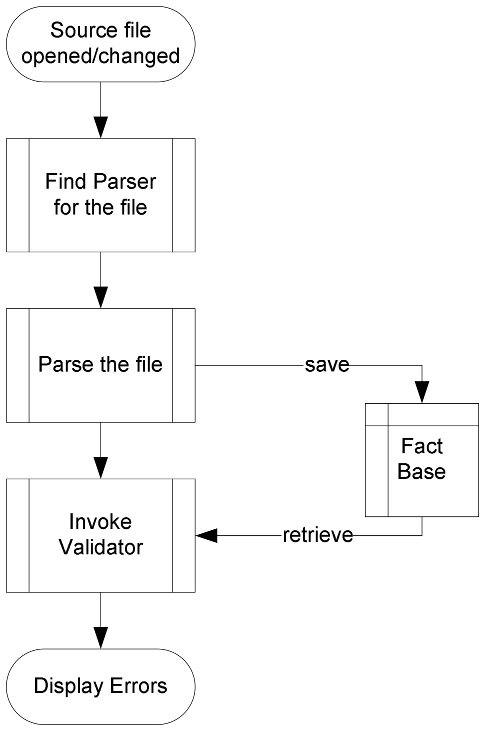


Figure 17: Crystallization Workflow

The crystallization framework consists of two parts: First, there is the visualization component responsible for visualizing dependencies. Valid and invalid dependencies are presented using the existing style available in Eclipse. Second, there are the parser and validator that understand a specific technology. The visualization component relies on the parser and validator to extract and validate dependencies. Dependency information such as origin, destination, and validity, are passed back to the visualization component for presentation purposes.

The crystallization framework is responsible for presenting dynamic dependencies in a manner that does not interfere with ongoing programming activities. Our integration of crystallization into Eclipse does not clutter the existing Java Development Tool (JDT) user interface. Dependencies are presented to developers as HTML like links that become visible only when the “control” key is pressed while the curser is over the origin of a dynamic dependency. By clicking on the link, the IDE unveils the implementing source code in an editor. This allows the developer to easily comprehend dynamic dependencies. The framework displays erroneous dependencies by placing problem markers, shown as red crosses, beside their origins in the source code editor.

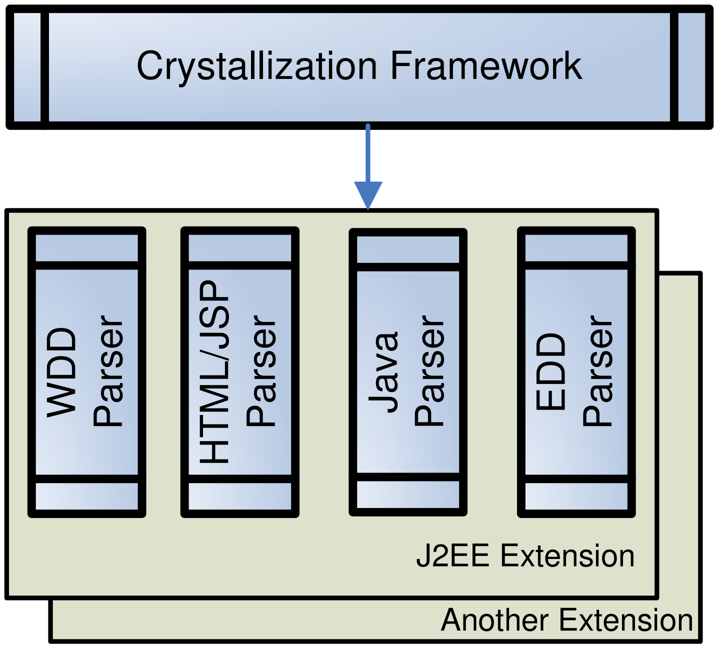


Figure 18: Crystallization Framework Architecture

As shown in Figure 18, the Crystallization Framework relies on *extensions* [1] to detect and validate dynamic dependencies. The crystallization framework is implemented as an Eclipse *extension point* [1] to leverage the capability of extension automatic discovery. A crystallization extension point contains information of the name of the extractor and the class implementing the extractor, and the name of the validator and the class implementing the validator. The following is a snippet of the crystallization extension point schema:

```
<?xml version='1.0' encoding='UTF-8'?>
<schema targetNamespace="swagkit">
…
   <meta.schema plugin="swagkit" id="crystallization" />
   <element name="parser">
         <attribute name="class" type="string">
         <meta.attribute kind="java"/>
   <element name="validator">
         <attribute name="class" type="string">
         <meta.attribute kind="java"/>
   </element>
</schema>
```

This schema dictates the structure of the extension. The line `<meta.schema plugin="swagkit" id="crystallization" />` contains the name of the plugin, e.g., *swagkit*, defining the extension point and the id of the extension point, e.g., *crystallization*. The global identifier consists of the name of the plugin and the id, e.g. *swagkig.crystallization*. The extension point contains two more elements, the parser, and the validator. Each of them must have a *string* indicating the name of the *java class* that implements the extension point.

The schema dictates the structure of the xml that is used to register new extensions. The following is a snippet of the registration xml file that registers an extension called "*J2EE Servlet*". Both the *parser* and *validator* element contains a *name* attribute, the name of the extension, and a *class* attribute, the implementing Java class.

```
<extension
         id="j2eeservlet"
         name="J2EE Servlet"
         point="swagkit.crystallization">
         <parser name="Servlet Parser"
class="edu.uwaterloo.swag.extractor.impl.compilationunit.ServletR
eferenceExtractor"/>
```

```
            <validator name="Servlet Dependency Validator"
class="edu.uwaterloo.swag.extractor.impl.compilationunit.ServletR
eferenceVerifier"/>
</extension>
```

All extensions must implement a *parser* that extracts a specific type of dependency and a *validator* that validates the extracted dependency. When Eclipse starts, it discovers all the installed extensions by parsing the above xml. No object instance is created until the crystallization framework needs it. As shown in the above sample extension, , the `ServletReferenceExtractor` is the parser and `ServletReferenceVerifier` is the validator of the new extension. `ServletReferenceExtractor` is instantiated when crystallization tries to parse an opened Servlet file to extract all Servlet related facts. `ServletReferenceVerifier` will be used by the visualization component to verify the extracted dependencies.

Although we have only implemented a J2EE extension to assist developers in comprehending J2EE applications, the crystallization framework is extensible to cover other technologies in J2EE, application frameworks such as Microsoft .NET[25], and add-on frameworks such as Struts[24].

4.5 Crystallization Extension Installation

Installation of crystallization extensions is straight forward. Simply adding the compiled binary code of the extension into a predefined directory together with the metadata such as the extension registration xml file will make the extension available to the Eclipse runtime. Eclipse even supports dynamic enabling of newly installed extensions without restarting Eclipse. When Eclipse starts, it searches for the predefined directories for existing plug-ins, which may or may not contain extensions. Eclipse also parses the metadata contained in each plug-in for information such as the location of the Java classes, etc. The Java classes are not instantiated until they are actually used.

4.6 Dependency Visualization

One of the responsibilities of the crystallization framework is to present the dependencies in a way that is convenient to the developer. The presentation of the dependency should not interfere with the ongoing activities of developers. For example, we do not want to display all the recovered dependencies since this would confuse the user interface with information that is not entirely pertinent. However, the presentation of the dependency information should be readily available to the developer, especially when developers change from programming to debugging.

Besides revealing valid and erroneous dynamic dependencies using HTML like links and problem markers, we have also implemented a class view that graphically presents all dynamic dependencies of the currently edited source file using arrows and boxes. The graph in the view contains the class itself and all the other application components that are referenced by syntactic or configuration relations. All components are represented as boxes and relations are represented as arrows. Valid and erroneous dynamic dependencies are differentiated using colors. This view provides developers with a high-level overview of all dynamic dependencies in the source file currently being edited.

We also leverage the existing JDT problem view by populating it with dependencies that failed the validation process. Invalid dependencies in the view can be sorted by different criteria such as severity, description, origin, etc. to allow the user to quickly search for erroneous dependencies.

4.7 Performance and Scalability

Reverse engineering is traditionally a slow process because it extracts and calculates dependencies from the complete code base. The steps of traditional reverse engineering include:

- Full scan of the code base to generate a fact base that is usually huge
- Rearrangement of the extract facts
- Visualizing extracted facts.

All the above steps are carried out based on the full system that is under examination. The time required to complete the above steps is proportional to the size of the application.

Since we are integrating our crystallization technique into Eclipse, it is unacceptable for this integration to incur a perceivable impact on its responsiveness. Even more importantly, the responsiveness should not be affected by the size of the application. Any slow down due to crystallization will seriously degrade the usability of the IDE. In order to achieve the required performance, we employ a "lazy" approach, which extracts and processes only those dependencies in the source code currently being edited. These extracted dependencies are cached into an in-memory database for reuse. Extracting facts from only source code currently being edited makes perfect sense since this is the context the developer is working in.

Since Crystallization extracts only dependencies from files that are currently being edited, the number of dependencies is reduced. Our extraction strategy ensures the ability to scale to large projects because a developer is only capable of working on a handful of files at any given moment. The number of files to be processed is not proportional to the size of the application either, instead; it is proportional to the number of files that a developer can work on concurrently.

The crystallization framework initiates a dependency extraction process whenever a source code file is opened in the editor or changed. In one test, Eclipse demonstrated acceptable responsiveness with one million dependencies in the in-memory database. When editing a Servlet with 898 lines of code, invalid dependencies are recovered without noticeable delay.

In this chapter, we introduced an extensible crystallization framework that can be extended to recover dependencies built upon various application frameworks. It can also be extended to extract dependencies from applications that are build upon add-on frameworks such as *struts*[24]. The *parsers* and *validators* that are contributed by extensions are invoked on demand to reduce the impact on the responsiveness of the IDE.

5 Case Study

In this chapter, we describe validation that we have done to verify the proposed and implemented crystallization framework.

5.1 The Pet Store Application

The Pet Store application is a sample J2EE application [3] from Sun Microsystems used to evangelize J2EE technologies. The Pet Store application demonstrates the capabilities these technologies provide to develop robust, scalable, portable and maintainable distributed e-business enterprise applications. The Pet Store application is a good candidate for our case study not only because it covers all the technologies that are provided in the J2EE specification, but also because it is a mid-size application. The following table enumerates the artifacts found in the Pet Store application.

Number of Java Classes	283
Number of JSPs	75
Lines of Java Source Code	45261
EJB Components	23
Lines of Configuration (WDD + EDD)	14710
Other Configuration Files	20

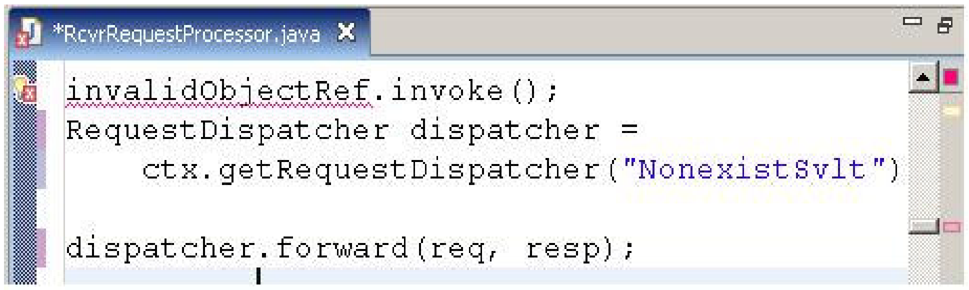
We use this application to demonstrate how the crystallization technique increases the visibility of erroneous dependencies and hence improves programming productivity, e.g. both efficiency and quality.

We imported the source code of the Pet Store into an Eclipse project. There were numerous compilation errors detected before we correctly configured the classpath and source folders of the project. These errors disappeared after the classpath of the project contained the required library jar files and source folders. When a class is successfully compiled, the Java editor allows developers to traverse the syntactic relationships such as class references, method invocations by clicking on the name of the class or method. But there is no traversal support for references to Servlets, JSPs[1], etc. Clicking on EJB remote interfaces will open the remote interface itself in the Java editor. Using the built-in search function of JDT for the implementation of the remote interface will not reveal anything since the implementation is not recommended to implement (inherit) the remote interface.

5.2 Dependency Manifestation

The editor contained in the unmodified Eclipse provides real-time dependency checking and syntax highlighting. Eclipse pinpoints erroneous static dependencies though the use of problem markers and underlining. Within the Crystallization Framework, invalid dynamic dependencies are shown in the same manner.

[1] The Web Tools Platform project [21] has tools that support traversal of relationship between JSPs, but, there is still no support for traversal of relationship involving Servlets. It does support reference to JSPs from Servlets either.

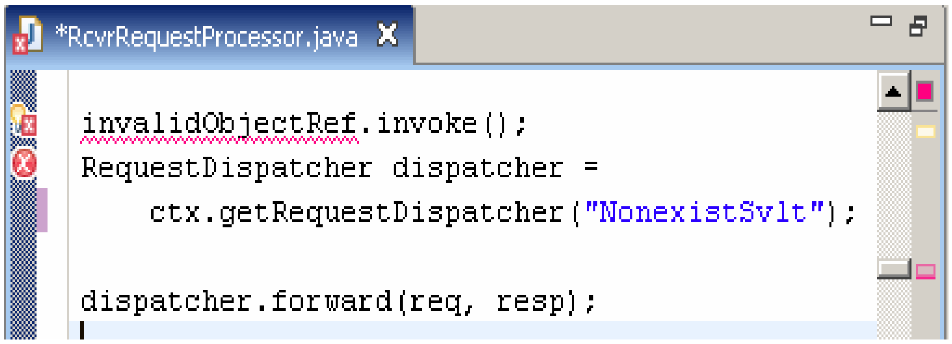


*RcvrRequestProcessor.java

```
invalidObjectRef.invoke();
RequestDispatcher dispatcher =
    ctx.getRequestDispatcher("NonexistSvlt")

dispatcher.forward(req, resp);
```

Figure 19: Highlighting Invalid Dependency in Unmodified Eclipse

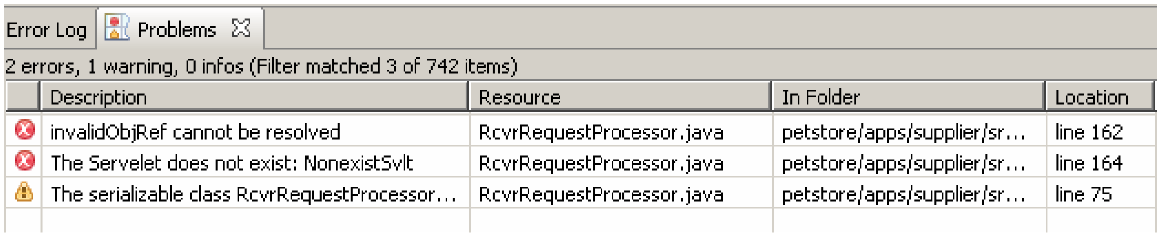


*RcvrRequestProcessor.java

```
invalidObjectRef.invoke();
RequestDispatcher dispatcher =
    ctx.getRequestDispatcher("NonexistSvlt");

dispatcher.forward(req, resp);
```

Figure 20: Highlighting Invalid Dependency in "Crystallized" Eclipse



Error Log | Problems

2 errors, 1 warning, 0 infos (Filter matched 3 of 742 items)

	Description	Resource	In Folder	Location
⊗	invalidObjRef cannot be resolved	RcvrRequestProcessor.java	petstore/apps/supplier/sr...	line 162
⊗	The Servelet does not exist: NonexistSvlt	RcvrRequestProcessor.java	petstore/apps/supplier/sr...	line 164
⚠	The serializable class RcvrRequestProcessor...	RcvrRequestProcessor.java	petstore/apps/supplier/sr...	line 75

Figure 21: Highlighting Invalid Dependency in Problems View

Figure 19 is a screen shot of the unmodified Eclipse showing an invalid dependency, "invalidObjectRef". The "Problems" view, as shown above, is the view that displays all problems that are recovered by Eclipse. The invalid dependency, a reference to "NonexistSvlt", is not displayed in the view. However, the "crystallized" Eclipse is aware of both the "invalidObjectRef" and the "NonexistSvlt" problem, as shown in Figure 20.

The invalid dependency is shown on the left hand side of the editor pane as a cross. Invalid Servlet references are shown in the same manner as invalid object references.

Figure 21 is a screen shot of the problems view which contains both syntactic and configuration dependencies. Each problem in the view has a description, the resource containing the problem, and the line number of the problem. This view allows the user to sort all of the “problems” found in the file opened by the currently active editor.

The attention grabbing red color used in the “cross” enables developers to quickly identify problems. On the right hand side is a bookmark, which if clicked will take the developer immediately to the origin of the invalid dependency. This is especially beneficial when working with large source files.

The “lazy” approach we employ allows us to focus on a specific source file without incurring a noticeable impact on responsiveness. Dependencies are extracted from the source code and deployment descriptors and validated on the fly as the developer is typing. Changes to source files trigger the extraction and validation process to ensure the up-to-date analysis of dependencies.

Since extracted dependencies are not discarded when developers switch to another source file, we do not perceive any negative impact in responsiveness after many files are scrutinized. This ensures the scalability of the enhanced Eclipse.

5.3 Source File Visualization

Although the erroneous dependencies are indicated clearly in the source editor and the problems view, this presentation is insufficient to provide a concise overview of large source files with many dependencies. We have thus introduced a “Class View” which presents dynamic dependencies that are extracted by our crystallization process.

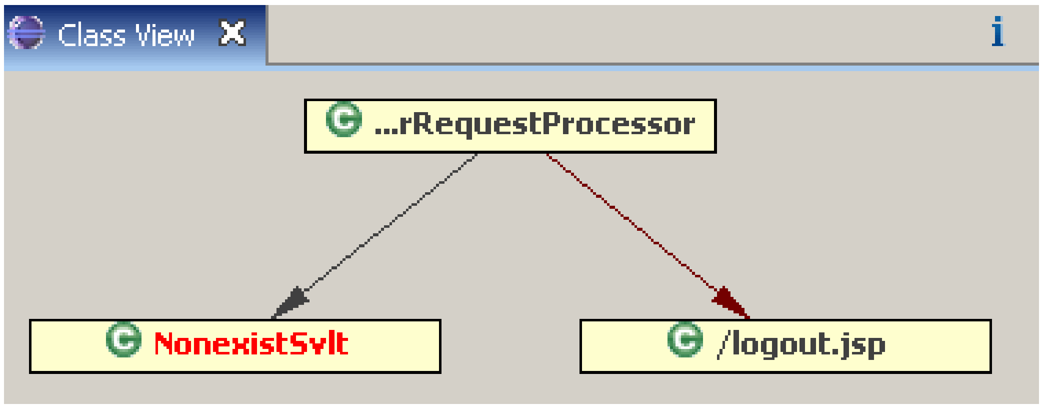


Figure 22: Class View

Figure 22 shows the class viewer of dynamic dependencies extracted from the “RcvrRequestProcessor” servlet. It shows a dependency against the “NonexistSvlt” in red which indicates an erroneous dependency. This view is refreshed whenever there is a change to the source code.

5.4 Summary of Case Study

The crystallization process has been successfully applied to the mid-sized J2EE application, Pet Store. The extensions that have been built are able to extract and validate pertinent dependency information. The visualization provided by our implementation was also found helpful in problem diagnosis. In cases where literal strings are used to reference J2EE components including web components and JMS queues, the crystallization framework is able to detect component references and validate them. Although there are about 70% of the references, mostly web component references that are using literal strings, there are still 28% of the references that are using constants. There are also places that the name of the components is calculated at runtime based on project specific naming conventions. Although the current crystallization extensions are not able to tackle component references using contants, we have identified ways to improve the extensions. It is very important to have the capability to detect references using constants because the best practices suggest using contants to reference components other than web components.

The crystallization extension was installed on 20+ developers to help them gain deeper and broader understanding of a large application with more than 300 modules and 200 megabytes of source code. The extension did not incur any noticeable slowed in the IDE. Since developers no longer need to go through hundreds of directories to find configuration files and to verify the configuration, the time for problem diagnosis was reduced from minutes, sometimes hours, to seconds.

6 Conclusion and Future Work

6.1 Conclusion

In this thesis, we have shown that with the capability provided by the crystallization framework, developers can obtain prompt feedback about not only syntactic validation, but also configuration verification. With our integrated development environment, the validation is carried out in the background without user intervention. The IDE helps developers to create "correct" application with a smaller number of "trial and error" cycles. A programming language should not be considered "good" without a IDE that helps improving the correctness of developed application.

Typically developers use a "trial and error" strategy to gain program comprehension and to test applications. Feedback from IDEs assists developers in understanding the dependencies between components. The speed of the feedback is one of the primary contributing factors affecting not only effectiveness of program comprehension, but also the efficiency of software development. Without prompt feedback on dynamic dependencies, the "trial and error" learning cycle is prolonged and hence results in a longer learning cycle. Developers must either wait until runtime or manually inspect the configuration and source code to observe dynamic dependencies. This resulting long turn around time complicates the program comprehension process.

The context sensitive approach the crystallization process takes helps developers focus on the current work and places developers in the current configuration context when validating the program under construction. Only source files that are opened by developers are validated. In another word, the crystallization framework uses the opened files as the scope of recovery.

Crystallization allows IDEs to detect and validate dynamic dependencies in application and to report the results to the developers. The results of crystallization can potentially improve developer efficiency, coding productivity and code quality.

6.2 Future Work

There is considerable amount of work that we were not able to complete due to time and resource constraints. First, the case study we carried out was based on an intermediate sized application. It would be a further validation of our technique if it was carried out on a larger sized enterprise application. Second, ideally a case study could compare the development productivity of two groups. One group would use the unmodified version of Eclipse and the other would use our enhanced Eclipse. Comparing the time to develop an application and the number of defects found would be valuable metrics to measure the effectiveness of the crystallization process. Another technique that could be used to evaluate the crystallization process is to measure the total development time spent to reach the same level of quality.

Our implementation covers only the cases where string literals are used to represent component names. Consequently, it is not able to detect dependencies on components referenced using constant string variables, even though their values are known at compile time. Further analysis of the Java Abstract Syntax Tree should allow us to recover names of components that are referenced using string constants.

There are other J2EE technologies that need to be further investigated in order to completely crystallize other types of J2EE applications. Although our research covers several important areas of J2EE, we did not crystallize tag libraries. Tag library technology is one the most important technologies that simplifies Java Server Page development. It allows developers to define custom tags similar to HTML tags. These tags are mapped to a Java class executed on the J2EE server and generate HTML code. An HTML or JSP page using a custom tag has a dependency on the tag library's implementing Java class which is specified in the tag library configuration file.

We have not used extracted dynamic dependencies to help derive application architectures. With an integrated application architecture viewer, the crystallization process can assist developers, especially newcomers to grasp the intricacies of application components and their interrelations and thus gain program comprehension.

Bibliography:

[1] The Eclipse Project, http://www.eclipse.org

[2] R. C. Holt. An Introduction to TA: the Tuple-Attribute Language, March 1997

[3] Java Pet Store. http://java.sun.com/developer/releases/petstore/, Sun Microsystems Inc.

[4] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Muller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf, *IBM Systems Journal,* Vol. 36, No. 4, pp. 564-593, November 1997.

[5] Inversion of Control Containers and the Dependency Injection Pattern. Martin Fowler, http://www.martinfowler.com/articles/injection.html

[6] Ahmed E. Hassan. Architecture Recovery of Web Applications, *Master's Thesis. Department of Computer Science, Faculty of Mathematics, University of Waterloo, Ontario, Canada. 2001*

[7] JavaServer Pages Technology - Documentation. http://java.sun.com/products/jsp/docs.html

[8] Enterprise JavaBeans Fundamentals: Introduction.
http://java.sun.com/developer/onlineTraining/EJBIntro/

[9] J2EE introduction. http://java.sun.com/developer/technicalArticles/J2EE/Intro/

[10] Lei Wu, Houari Sahraoui, Petki Valtchev. Program comprehension with dynamic recovery of code collaboration patterns and roles. *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research. 2004*

[11] Kenny Wong. Software Understanding through integrated structural and run-time analysis. *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research.* 1994.

[12] Carlo Bellettini, Alessandro Marchetoo, Andrea Trentini. WebUml: Reverse Engineering of Web Applications. *Proceedings of the 2004 ACM symposium on Applied computing. 2004.*

[13] Eleni Stroulia, Tarja Systä. Dynamic Analysis for Reverse Engineering and Program Understanding. *ACM SIGAPP Applied Computing Review. 2002.*

[14] Basili, V. R., Green, S., Laitenberger, O., Shull, F., Sørumgård, S., and Zelkowitz, M. V. 1995 *The Empirical Investigation of Perspective-Based Reading*. Technical Report. UMI Order Number: CS-TR-3585., University of Maryland at College Park.

[15] H. Müller, K. Wong, and S. Tilley, "*Understanding software systems using reverse engineering technology*", In The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS), 1994.

[16] HyperText Markup Language (HTML) Home Page, http://www.w3.org/MarkUp/

[17] Uniform Resource Locators (URL), http://www.ietf.org/rfc/rfc1738.txt

[18] LSEdit, http://www.swag.uwaterloo.ca/lsedit/index.html

[19] H. Muller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[20] Case, A. F. 1985. Computer-aided software engineering (CASE): technology for improving software development productivity. *SIGMIS Database* 17, 1 (Sep. 1985), 35-43. DOI= http://doi.acm.org/10.1145/1040694.1040698

[21] Eclipse WTP Project, http://www.eclipse.org/webtools/

[22] HTTP - Hypertext Transfer Protocol, http://www.w3.org/Protocols/

[23] Introduction to Grok,
http://swag.uwaterloo.ca/~nsynytskyy/grokdoc/grokintro.html

[24] Apache Struts Project, http://struts.apache.org/

[25] Microsoft .NET, http://www.microsoft.com/net/default.mspx

[26] Jerding, Dean F. and Stasko, John T., " Using Visualization to Foster Object-Oriented Program Understanding", Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Technical Report GIT-GVU-94-33, July 1994.

[27] OptimalJ, http://www.compuware.com/products/optimalj/

[28] IBM Rational Software, www.ibm.com/software/**rational**

[29] NetBeans, http://www.netbeans.org/

[30] IntelliJ IDEA, http://www.jetbrains.com/idea/