Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Dependency Injection with Static Analysis and Context-Aware Policy

Michael D. Ekstrand[a] Michael Ludwig[b]

a. Dept. of Computer Science, Texas State University
http://cs.txstate.edu/

b. Dept. of Computer Science and Engineering, University of Minnesota
http://www.cs.umn.edu/

Abstract The dependency injection design pattern improves the configurability, testability, and maintainability of object-oriented applications by decoupling components from both the concrete implementations of their dependencies and the strategy employed to select those implementations. In recent years, a number of libraries have emerged that provide automated support for constructing and connecting dependency-injected objects. Our experience developing systems with these tools has led us to identify two shortcomings of existing dependency injection solutions: the mechanisms for specifying component implementations often make it difficult to write and configure systems of arbitrarily-composable components, and the toolkit implementations often provide limited capabilities for inspection and static analysis of the object graphs of dependency-injected systems. We present Grapht, an new dependency injection container for Java that addresses these issues by providing *context-aware policy*, allowing component implementation decisions to depend on where in the object graph a component is required, and using a design that allows for static analysis of configured object graphs. To achieve its objectives, Grapht is built on a mathematical representation of dependency injection and object graphs that facilitates static analysis and straightforward implementation, and forms a basis for further consideration of the capabilities of dependency injection. The mathematical representation includes context-aware policy that we show to be strictly more expressive than the qualified dependencies used in many current toolkits. We demonstrate the utility of our approach with a case study showing how Grapht has aided in the development of the LensKit recommender systems toolkit.

Keywords dependency injection, component instantiation, Java

1 Introduction

Large software systems often comprise many components that work in concert to provide the system's functionality. In the last twenty years, *dependency injection* (DI) [Fow04, YTM08] has gained prominence as a design pattern for organizing the components of such a system, particularly in object-oriented programming environments.

If a component A requires another component B in order to fulfill its obligations, there are several ways that it can obtain a suitable reference. A can instantiate B directly; this is straightforward, but makes it difficult to substitute alternative implementations of B. A can obtain B from some other service, like a factory or service locator, allowing alternative implementations to be used but making A dependent on the resolution strategy. Finally, in dependency injection, A can require B to be provided via a constructor argument. That is, the dependency (B) is *injected* into A. Whatever component creates A is therefore free to substitute an alternate implementation or reconfigure B in any way it wishes. Listing 1 shows a simple component with a single dependency written in both a classical direct instantiation style (a) and using the dependency injection pattern (b).

When used throughout the design of a system, dependency injection provides a number of advantages, mostly following from reduced coupling between components. Some of these benefits include:

- Components are free of all knowledge of the implementations of their dependencies — they do not even know what classes implement them or how to instantiate them, only that they will be provided with a component implementing the interface they require.
- Components can be reconfigured by changing the implementations of their dependencies without any modification to the components themselves.
- Components can be more easily tested by substituting mock implementations of their dependencies. While mocking is not new, the component design encouraged by dependency injection makes it particularly easy to substitute mock objects.
- Each component's dependencies are explicit, appearing as formal arguments of the constructor and initialization methods, so the component's interaction with the rest of the system is largely self-documenting. This can make the system easier to understand and more amenable to static analysis.

```
public class Component {
    private Dependency dependency;

    public Component() {
        dep = new DependencyImpl();
    }
}
```

(a) Direct instantiation.

```
public class Component {
    private Dependency dependency;

    @Inject
    public Component(Dependency dep) {
        dependency = dep;
    }
}
```

(b) Dependency injection.

Listing 1. A simple component with and without dependency injection.

The reduced coupling and increased flexibility of dependency injection comes with a cost: in order to instantiate a component, its dependencies must be instantiated first and routed through a myriad of constructor parameters. This requires the code initializing a component to know its dependencies, construct them in the proper order, and pass them in to the constructor. Doing this manually, while possible, is cumbersome. Several systems, called *dependency injection containers*, have been developed to provide automated support for configuring and instantiating the graphs of objects that arise when building applications with dependency injection. The Java community has developed a number of standards, most notably JSR 330 [JL09], that specify common behavior for such tools.

We have made extensive use of dependency injection in developing several software projects, including the LensKit recommender systems toolkit [ELKR11, Eks14], the MovieLens movie recommender web site[1], and a book recommendation web service for libraries called BookLens[2]. Through this development experience, we have discovered two significant shortcomings in the available tools to support dependency injection:

Static Analysis Most current dependency injectors do not form an instantiation plan prior to instantiation, instead resolving dependencies lazily as needed. Without such a plan, the types of and efficiency of static analysis is severely limited, if not impossible.

Composability The available means to configure dependency-injected applications often place limits on the ways in which components can be composed. Most frameworks do not deal well with the same class or interface appearing in multiple places in the object graph with different implementations or configurations, and the tools they do provide for such scenarios are weak.

To address these issues, we have developed a language-independent, graph-based model of dependency injection. This model serves as the basis for static analysis of components in dependency-injected systems and is amenable to context-sensitive configuration policy, easeing the configuration of complex configurations of arbitrarily composable components. We have used this model to build a dependency injection toolkit for the Java platform, dubbed 'Grapht'. Finally, we have used Grapht to build several static analysis features for the LensKit toolkit, and found that the ease of expressing context-sensitive configurations leads to solutions to some class design and configuration issues that have certain benefits over the solutions admitted by current technologies. We also use our formal model to show that context-sensitive policy is strictly more expressive than the configuration capabilities provided by common JSR 330-compliant containers.

Our contributions are as follows:

1. A survey of potential requirements for dependency injection containers, and the ability of existing containers to meet those requirements.

2. A mathematical formalization of the dependency injection problem and graph-based model of solutions to it.

3. A description and model of the idea of context-sensitive policy, enabling a greater degree of composability than common existing solutions, and a proof that it is strictly more expressive than previous mechanisms.

[1] `http://movielens.org`
[2] `http://booklens.umn.edu`

4. A Java implementation of a DI container built on the model in (2), with support for static analysis of component graphs.
5. A discussion of the implications of easy expression of context-sensitive component configuration for the design of object-oriented software.

In this paper, we first describe in more detail the operation and requirements of a DI container and survey related research and software projects. We then describe our mathematical model of dependency injection, discuss how it is implemented in Grapht, and conclude by describing several techniques and analyses made possible with Grapht that greatly simplify complex application development.

2 Related Work

Despite widespread use of dependency injection by the software development community, we have been able to find little treatment of the subject in the research literature. Yang *et al.* [YTM08] empirically studied the prevalence of its use, and Razina and Janzen [RJ07] studied its impact on maintainability measures such as coupling and cohesion. DI has also been shown to be effective for configuring game components [PSC$^+$10] and connected with aspect-oriented programming [CI05], but there does not seem to be much other published research on dependency injection since Fowler [Fow04] provided its modern formulation. Prasanna [Pra09] discusses extensively the core principles of dependency injection and support for it in the Java platform as provided by Guice and Spring, but there has been little treatment of models for reasoning about its capabilities, limitations, and potential extensions. Hudli and Hudli [HH13] provide some static verification of dependency injection, but their verification is limited to checking the type-safety and completeness of injected components.

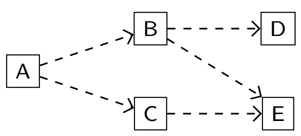
Some of the literature on component instantiation anticipated aspects of dependency injection. Magee *et al.* [MDEK95] describe the Darwin notation for describing software component wiring; it is formalized in terms of the π-calculus, and seems amenable to static analysis, but required the entire system component graph to be explicitly specified.

Oustide of published literature literature, there are a number of software tools to provide automated support for dependency injection. We discuss the these tools more thoroughly in sections 3.1 and 3.8.

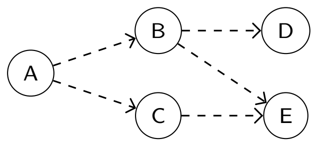
3 Operation of a DI Container

The task of a dependency injection container (sometimes called a dependency injector) is to instantiate and connect a graph of objects, each designed with the dependency injection pattern, that will together provide the application's functionality. This often takes the form of instantiating a root component along with its dependencies. Figure 1 shows an example; the DI container is responsible for transforming class A along with its dependencies (fig. 1a) into a graph of objects (fig. 1b). In a web application, for example, the root will often be a request dispatcher or the handler class for a particular request, allowing an application configuration to map URLs to handler classes and rely on the DI container to make sure each handler class has all its dependencies.

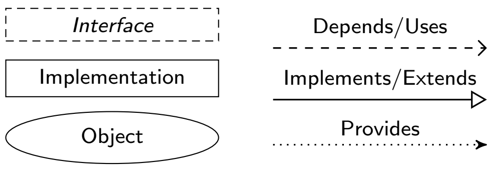
To accomplish this task, the container must typically do three things:



(a) Class diagram



(b) Object diagram



(c) Shape legend, containing the shapes used in class and object diagrams throughout this paper.

Figure 1. Class dependencies, object graph, and legend.

1. Identify the dependencies of each component, using reflection, source code analysis, or some other mechanism appropriate to the programming environment.

2. Find an appropriate implementation for each dependency according to the *policy* specified by the application's configuration. This policy may be altered to adapt the application to different deployment environments, configurations, or other requirements.

3. Instantiate the final object graph, instantiating each individual component (as appropriate) and wiring them together to ensure all dependencies are satisfied.

These tasks can be performed together, identifying and resolving dependencies lazily in response to object instantiation requests, or with a phased approach in which a dependency solution or instantiation plan is computed as an object in its own right and passed to a separate component to perform instantiation. To enable static analysis of the object graph, we take the multi-phase approach; in any case, considering the tasks separately makes it easier to describe the required behavior of a container.

Tasks 1 and 3 are highly dependent on, and largely prescribed by, the design and capabilities of the programming environment in which dependency injection is being performed (e.g. the Java virtual machine, .NET runtime, Python interpreter, or a Standard ML code generator). The remainder of this section is concerned primarily with the second task, describing several types of dependency relationships that arise and considerations for resolving and configuring them.

A number of the examples we use are drawn from our experience building and using the LensKit toolkit [ELKR11, Eks14]. LensKit is an open-source toolkit that supports research and prototyping of *recommender systems*, artificial intelligence tools for recommending items for users to purchase, read, or otherwise consume. Dependency injection has enabled us to build flexible and extensively reconfigurable implementations of common recommender algorithms. Algorithms are built up of many individual components representing filters, data sources, prediction algorithms, and more; one such component is the *ItemScorer*, responsible for scoring items by estimating how much a user is likely to like a particular item. Developers can select component implementations and parameter values to alter the recommender's behavior and optimize it for their application, and researchers can implement new components and reuse existing ones to experiment with new recommendation ideas.

3.1 Existing Tools

Before describing the general operation and requirements of a DI container, it is useful to lay out the ones currently available for the Java platform. JSR 330 [JL09] standardizes dependency annotations and behavior for DI containers in Java, and most Java DI containers are adding JSR 330 support if they do not have it already. JSR 330 itself is based heavily on the design of Google's Guice DI container [LBW09]; Spring [Joh02] and PicoContainer [Ham11] are also used significantly in Java applications and implement JSR 330. These three are the most widely-used injection toolkits for Java.

Guice and PicoContainer are broadly similar systems. Both provide a Java API to specify dependency configuration and lazily resolve dependencies while instantiating objects (with the consequence that there is no pre-computed dependency plan). They provide hooks that can intercept and record the dependency solution as it is being discovered, and tools exist that use this to produce dependency graphs. Guice also supports extensive defaulting (called *just-in-time* injection), looking up default implementations from Java annotations; PicoContainer requires the application to explicitly describe all component implementations that will participate in injection.

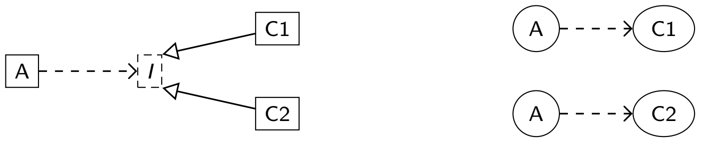
Spring is an expansive application framework, providing tools for web development, aspect-oriented programming, and dependency injection, among other things. Early versions of Spring used XML descriptions of the complete object graph to describe the implementations to use. Although verbose, this is a powerful way of configuring dependencies and can achieve the same results as context-aware configuration. Spring has more recently been updated to support the JSR 330 annotations and automatically configure certain types.

A newer entrant, Dagger [Wil12], is a lighter DI container that makes extensive use of static analysis of application source code. It is heavily inspired by Guice, but is particularly geared towards the Android platform.

Outside the Java ecosystem, Ninject [Koh12] provides similar DI services for .NET. Ninject supports both just-in-time binding and context-aware binding with an elegant API that avoids the verbosity of Spring's context-sensitive solutions. In Ninject, bindings are provided with the injection context and can invoke an arbitrary boolean function to determine if it matches. When instantiating or 'activating' objects, Ninject proceeds in a lazy fashion like Guice, allocating new instances as necessary to satisfy the parameters of a required instance. As far as we know, Ninject does not provide any support for static analysis or manipulation of object graphs.

Besides Ninject, .NET has several other DI or frameworks, including The Castle Windsor[3], StructureMap[4], AutoFac[5], and Unity[6]. These provide very similar configuration and binding capabilities as the Java frameworks surveyed above and operate at runtime using C#'s built-in reflection.

[3]`http://docs.castleproject.org/Windsor.MainPage.ashx`
[4]`http://docs.structuremap.net`
[5]`http://autofac.org`
[6]`https://unity.codeplex.com`



(a) Class diagram (b) Potential object graphs

Figure 2. Interface with multiple implementations.

3.2 Basic Policy

In order to instantiate an object graph, the DI container must know what classes should be instantiated to satisfy each dependency. For example, to instantiate a LensKit recommender, it needs to know what implementation of *ItemScorer* should be used; for a web application, it needs to know what request dispatcher to instantiate and how to configure its dependencies.

If all the dependencies in question are concrete classes (as in Fig. 1a) or the runtime environment has no concept of interfaces or polymorphism, then resolving dependencies is simply a matter of looking up the constructor for each dependency.

If there are multiple implementations of component interfaces to choose from, the DI container needs some form of *dependency policy* [Mar96] to determine which implementation to use to satisfy each dependency. Figure 2 shows a simple class diagram where a dependency policy is necessary; interface *I* has two implementations C1 and C2. When instantiating A, the injector needs to know which implementation to use to satisfy the dependency on *I*. Both graphs in fig. 2b are valid solutions, and which one is desired may depend on many factors.

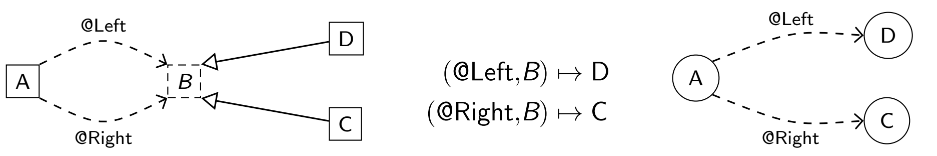
Policy can be provided by fully specifying the object graph, e.g. in an XML file. More commonly, however, it is defined by a more lightweight mechanism. One design, used by Guice and Grapht, uses *bindings* from component types (typically interfaces or abstract classes) to concrete classes implementing those types. To produce the top graph in fig. 2b from fig. 2a, the binding $I \mapsto \mathsf{C1}$ would be used; the binding $I \mapsto \mathsf{C2}$ produces the bottom graph.

In addition to binding interfaces to implementing classes, it is useful at times to bind them to pre-instantiated objects or to *providers*, factory components that can produce the desired object on demand. Finally, binding-based and specified-graph policy can be mixed, as a DI container could support binding an interface to a specified subgraph (which may itself have unresolved dependencies that are resolved via the binding mechanism).

One final source of dependency policy is defaults. In a language where interface definitions can be annotated, an interface can have an annotation specifying a default implementation to be used unless a more specific policy is provided. This works particularly well with binding-based policy to allow an application to largely wire itself, guided or overridden by a few key bindings.

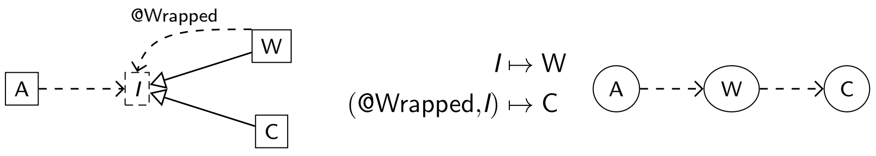
3.3 Qualifiers

Type information is not always sufficient to describe a component's dependencies. There are cases, such as that shown in fig. 3, where the same component interface is used in different roles and the desired configuration will use different implementations in each of these roles. To accommodate this, it is common to allow dependencies to



(a) Class diagram (b) Bindings (c) Example object graph

Figure 3. Component with qualified dependencies (indicated by edge labels).



(a) Class diagram (b) Bindings (c) Object graph

Figure 4. A wrapper component. W implements *I* by wrapping another component of type *I*, in this case C.

be annotated with tags (called *qualifiers* by JSR 330 and many tools, including ours; Prasanna [Pra09] addresses similar problems with *keys*) that indicate the role that dependency is intended to fill; in Java, qualifiers typically take the form of annotations applied to the dependencies. In fig. 3a, the component A has two dependencies of type *B* with the qualifiers @Left and @Right; in fig. 3c, each of these dependencies is satisfied with a different implementation.

Another use of qualifiers is to enable wrapper components to be created. As shown in fig. 4, these components implement an interface by wrapping another component of the same interface. The wrapper annotates its dependency on the wrapped component with a qualifier so the policy can distinguish between the primary binding (interface to wrapper) and the wrapped binding (qualified interface to implementation). LensKit makes extensive use of wrappers to allow data pre- and post-processing to be decoupled from more fundamental computation.

In LensKit, these two needs combine when we want to configure a hybrid component. We have an interface, *ItemScorer* (*IS*), that computes scores for items; a hybrid computes scores by blending the outputs of other scorers. Figure 5 shows the class and object diagrams for a simple 2-way hybrid that blends 'left' and 'right' scorers. Qualified dependencies and bindings allow this object graph to be configured easily.

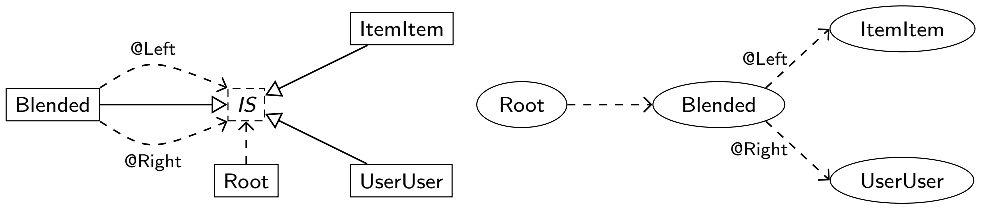


Figure 5. Hybrid item scorer in LensKit.

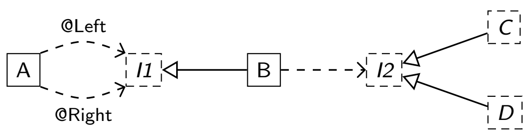
3.4 Context-Aware Policy

While qualifiers allow dependency policy to be conditional on annotations indicating how a dependency is going to be used, they offer limited composability in the context of larger object graphs. In fig. 6a, A has two qualified dependencies on *I1*. Unlike the case in fig. 3 where we want to use different implementations of *I1* for the left and right components, in this example we want to use the same class (B) with different implementations of dependency on *I2*. Qualifiers do not have sufficient expressiveness to allow two different configurations of B to be used as subgraphs of a larger object graph. Many attempted solutions via qualifiers or other mechanisms such as keys [Pra09, sec. 2.4], require the component to be aware of the contexts in which it may need to be configured; this impedes the ability to reuse and compose existing components in new ways. We see three ways that such graphs can be configured:

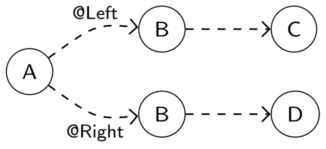
- Fully specify the layout of the object graph, or of some portion of it. This mechanism, supported by SPRING, has three shortcomings: the necessary configuration files are quite verbose, they result in configuration that is redundant with either defaults or other portions of the policy, and the configuration is more strongly affected by changes in the structure and relationships of the components.
- Bind the divergent components to pre-instantiated instances, or to providers that re-expose the dependencies with other qualifiers. This is the way to achieve such configurations in GUICE or PICOCONTAINER. Using providers results in an explosion of provider classes, one for each position in which a component might be needed. Both of these solutions impede static analysis, either by obscuring portions of the component graph by needless indirection through provider types or by requiring components to be instantiated outside the DI process (and therefore outside the realm of analysis).
- Provide a mechanism for specifying different policies to be employed at different points in the component graph (context-sensitive policy). Such a scheme requires less configuration, is amenable to static analysis, and is resilient to changes in the structure of the components to be configured so long as those changes do not affect the way in which the policy determines which rules to apply. These policies have the drawback of being more ‘magic’, making it less obvious how the policy will interact with the component definitions to produce a final component graph, particularly for readers less familiar with the underlying software. We are willing to make this a tradeoff in our applications, as additional explicit bindings can be added to make the configuration clearer and visualization tools can help with understanding configurations.

Context-sensitive policy can be represented by adding context to the bindings. A context-sensitive binding is only activated in certain portions of the object graph: in the case of fig. 6, the bindings for *I2* depend on whether they are being used to satisfy some (transitive) dependency of the left or the right B component. These bindings depend on the *context* of the dependency, a path through the dependency graph from the initially-requested component to the component whose dependency is to be satisfied.

We show in section 4.4 that qualifiers can be replaced with additional types and coupled with context-aware policy to achieve the same results as qualifier-based policy. Thus, context-sensitivity is a more general solution to the same set of problems as



(a) Class diagram



(b) Desired object graph

$$\vdash \mathit{I1} \mapsto \mathsf{B}$$
$$(\text{@Left},\mathsf{B}) \vdash \mathit{I2} \mapsto \mathsf{C}$$
$$(\text{@Right},\mathsf{B}) \vdash \mathit{I2} \mapsto \mathsf{D}$$

(c) Bindings

Figure 6. A dependency graph requiring context-aware policy. $X \vdash I \mapsto C$ denotes that I is bound to C only when satisfying dependencies of X.

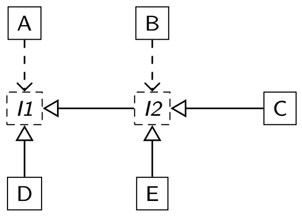


Figure 7. Class graph with subtypes

those solved by qualifiers. In practice, we continue to use qualifiers because they are specified by JSR 330 and because they are syntactically convenient.

3.5 Subtype Binding

Components are not limited to implementing a single interface. Interfaces can extend and refine other interfaces, and components can implement multiple distinct interfaces. The correct behavior in these situations is not necessarily obvious; we have used our experience building systems directly and with other DI containers and attempted to apply good judgment in resolving them. We have particularly sought to avoid allowing spurious and confusing bindings to arise from straightforward policy statements.

Consider the class diagram in fig. 7, where there are two interfaces and A depends on *I1*, while B depends on the more specific *I2*. For convenience and consistency with other containers, we would like the single binding $\mathit{I1} \mapsto C$, or even $\mathit{I2} \mapsto C$, to satisfy both dependencies. Requiring both bindings to be explicitly specified would increase both maintenance burden and the difficulty of determining exactly what binding rules are needed when building a configuration. However, either decision should be overrideable; if there are explicit bindings $\mathit{I1} \mapsto C$ and $\mathit{I2} \mapsto E$, C should not be used to satisfy B's dependency. Also, a binding $\mathit{I1} \mapsto D$ should clearly not be used satisfy B's dependency, as B requires an *I2*.

The particular policy we have adopted is as follows:

1. Explicit bindings always take priority.

2. The binding $A \mapsto \mathsf{B}$ should be treated as if it also bound every type that is a

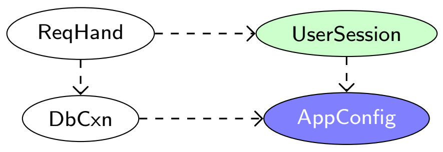


Figure 8. Web application with scopes

supertype of A, or a supertype of B and subtype of A, to B, unless such a binding (called a *generated* binding) conflicts with an explicit binding.

The behavior enabled by this policy is consistent with the common behavior of other DI containers, notably PicoContainer.

3.6 Scope Separation

In many applications, it is useful to have different *scopes* of objects. For example, fig. 8 shows some objects involved in handling a web request. The request handler (ReqHand) is instantiated for each request and is responsible for the primary dispatch and handling of that request. Each request gets its own database connection (DbCxn), likely obtained from a connection pool. The request also makes use of user session data (UserSession); this object is used to persist and retain state for the currently logged-in user, and needs to be shared between all requests servicing connections from that user. The global application configuration (AppConfig) is in yet another scope, being shared among all requests no matter their session (effectively a singleton).

In LensKit, we similarly have two major scopes: certain objects, such as statistical models, are immutable, thread-safe, and can be freely shared throughout the application; further, they are very costly to instantiate. If they require database access, they only need this access during the instantiation process; the final instantiated object can stand on its own. Other objects require access to a live database connection and, when LensKit is integrated into a web application, need to be created on a per-request basis.

Spring and Guice both provide support for scopes. Guice provides annotations that can be applied to classes, indicating the scope in which they should live; scope can also be configured in bindings. PicoContainer also supports scopes, but does so by using nested injection containers.

3.7 Static Analysis

We also desire the ability to conduct static analysis over component graphs. In this context, we consider an analysis to be static if it can be conducted over a graph representation of the components to be instantiated before actually instantiating any components (invoking constructors, providers, etc.), as opposed to a dynamic analysis that will operate by reflection on the component instances after they have been instantiated.

There are several things that we seek to accomplish with static analysis:

- Type-checking of components to ensure they satisfy the dependencies they are intended to satisfy (this analysis is provided by the static verification work of Hudli and Hudli [HH13]), allowing early detection of configuration errors.

- Scope or lifecycle analysis, examining a component and its dependencies to determine where it must be instantiated and how it can be shared between different applications, threads, or scopes. We use this in LensKit to automatically identify components that are 'shareable' (thread-safe and serializable) and, if they have no non-shareable dependencies, arrange for them to be pre-instantiated and shared between multiple recommender instances.
- Application-specific 'linting' to look for potentially erroneous or inefficient configurations. For example, in LensKit we warn the user if a shareable component cannot be shared due to dependency problems.
- Visualizations of component graphs and application configurations without requiring the component instantiation process to run in order to produce the visualizations.

Among the existing Java DI containers, Dagger stands out as making heavy use of static analysis. It uses Java annotation processors to validate an object graph and configuration at compile time, not just prior to instantiation in a running application launcher. It can also generate code to instantiate components instead of relying on reflection at runtime. However, its configuration API is limited compared to other tools because bindings and policies are specified solely by source code annotations.

3.8 Summary of Capabilities

Table 1 summarizes the Java container implementations we have discussed and the features they provide, including their support for the various requirements we have outlined. Although each of these is capable and useful as a general DI container, they all exhibit at least one of the shortcomings we have identified in our work on LensKit. We also note that Grapht does not completely implement all the features of other containers; to date, we have focused on developing the capabilities that LensKit requires. It is certainly feasible to implement many of the remaining features on top of Grapht, but it has not yet been a priority.

	Guice	Spring	PicoCont.	Dagger	Grapht
Graph Manipulation	✗	✗	✗	✗	✓
Static Analysis	✗	✗	✗	✓[a]	✓
Context-aware Policy	✗	Hard	✗	✗	✓
Just-in-Time Binding	✓	✓	✗	✓	✓
Scope Annotations	✓	✓	✗	✗	✗[b]
Circular Dependencies	✓	✓	✓	✗	✓[c]

[a]Validation and visualization only.
[b]Grapht has features that provide a good basis for supporting scope annotations, but we have not yet implemented this logic.
[c]While we omit circular dependencies from our model and discussion, the Grapht software supports them to achieve compatibility with JSR 330. Appendix A describes how this support works.

Table 1. Summary of DI containers

4 A Model of Dependency Injection

In this section, we present a formal model of dependency injection. This model is language-independent, so that we may consider the fundamental issues of dependency injection independent of any particular programming language or runtime environment in which the concept may be applied. While dependency injection is generally employed in object-oriented programming environments with some reflection capability, similar ideas can apply to other environments, including static, non-object-oriented systems such as the Standard ML module and functor language.

The core of our model is the *component request*, a request for an instance of some component type along with its required dependencies. Injector policy is realized by a *binding function* that is used to identify the correct implementation of the request and, recursively, to satisfy its dependencies. All of this happens within a *runtime environment*, a representation of the aspects of the programming environment relevant to dependency injection. Satisfying a component request with a well-formed policy results in a unique *constructor graph*.

This model has allowed us to reason about and define the specifics of GRAPHT's algorithms in concise terms without the encumbrance of the Java language and reflection system. We also use it to show that qualifiers are reducible to contexts and thus are strictly less expressive.

Our model and its presentation are organized as follows:

1. Define dependency injection component requests, solutions, and policy, with context and context-aware policy but omitting qualifiers.
2. Show how to solve DI component requests and produce good solution graphs.
3. Extend the definitions with qualifiers and show a reduction from qualified DI to unqualified but context-aware DI.

4.1 Core Definitions

Dependency injection occurs within the context of a particular *runtime environment*, provided by the language or platform's runtime facilities, libraries in use, and the running application's class definitions.

DEFINITION 1 (RUNTIME ENVIRONMENT)
A runtime environment $\mathcal{R}$ is a 6-tuple $(O, T, C, <:, \succ, \mathcal{D})$, where

- *O is the set of all possible objects.*
- *T is a non-empty set of types. In a dynamic language such as Python, we simply require some way for components to be labeled with a type; this could be a list of protocols that they implement, accompanied by a means of representing interfaces.*
- *C is a set of constructors.*
- *A subtyping relation $<:$ such that for two types $\tau, \tau' \in T$, $\tau <: \tau'$ means that τ is a subtype of τ'.*
- *A constructor-type relation $\succ \subset C \times T$, such that for $c \in C$ and $\tau \in T$, $c \succ \tau$ means that constructor c constructs objects of type τ. We require subtyping and constructor typing to be consistent, so if $c \succ \tau$ and $\tau <: \tau'$, then $c \succ \tau'$.*

$$\begin{tikzcd}
 & c_B \arrow[r, dashed, "\tau_D"] \arrow[rdd, dashed, "\tau_E"] & c_D \\
c_A \arrow[ru, dashed, "\tau_B"] \arrow[rd, dashed, "\tau_C"'] & & \\
 & c_C \arrow[r, dashed, "\tau_E"'] & c_E
\end{tikzcd}$$

Figure 9. Example constructor graph, corresponding to fig. 1a.

- *A dependency function $\mathcal{D} : C \to \mathcal{P}(T)$ such that $\mathcal{D}(c)$ represents the dependencies of a constructor $c \in C$.*

For the purposes of this model, constructors encapsulate any mechanism for object instantiation; setter and field injection is included in the concept, as are instances (represented by nullary constructors) and providers. For simplicity and clarity, we omit the notion of optional dependencies, but they do not impact the model.

The process of satisfying a component request results in a *constructor graph*:

Definition 2 (Constructor Graph)
A constructor graph *in a runtime environment $\mathcal{R}$ is a directed graph $G = (V, E)$ having*

- *a designated root vertex* $\textsc{Root}[G] \in V$
- *a constructor* $\mathrm{C}[v] \in C$ *associated with each vertex* $v \in V$
- *a type* $\textsc{Type}[e] \in T$ *labeling each edge* $e \in E$

The following properties must also hold:

1. *Each constructor produces the required type:*

$$\forall (v \xrightarrow{\tau} v') \in E . \mathrm{C}[v'] \succ \tau$$

2. *Each dependency set is fully satisfied with no extraneous dependencies:*

$$\forall v \in V . \{\tau \in T : (v \xrightarrow{\tau} v') \in E\} = \mathcal{D}(\mathrm{C}[v])$$

Property (1) requires that the constructor graph is type-safe (no constructor will produces a type that is incompatible with the dependency it is intended to satisfy), and property (2) requires that every dependency is satisfied, and there are no stray edges to clutter up the graph (every edge is actually used).

Figure 9 shows an example constructor graph, corresponding to the class diagram in fig. 1a.

Further, it is useful to define a notion of equality between constructor graphs.

Definition 3 (Constructor Graph Equality)
Two constructor (sub-)graphs of $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ rooted at v_1 and v_2 are equal if the following hold:

- $\mathrm{C}[v_1] = \mathrm{C}[v_2]$
- *For each $\tau \in \mathcal{D}(\mathrm{C}[v_1])$, there exist $v_1' \in V_1$ and $v_2' \in V_2$ such that $v_1 \xrightarrow{\tau} v_1' \in E_1$, $v_2 \xrightarrow{\tau} v_2' \in E_2$, and the subgraphs rooted at v_1' and v_2' are equal.*

In order to support context-aware dependency policy, we define a *context*:

Definition 4 (Context)
A context $\chi = \langle c_1, \dots, c_n \rangle$ is a finite sequence of constructors, a path from the root of a constructor graph G ($c_1 = \mathrm{C}[\textsc{Root}[G]]$).

The set of all contexts is denoted X; the empty context is $\langle\rangle$. The concatenation of two contexts χ_1 and χ_2 is denoted $\chi_1 +\!\!+ \chi_2$.

With these foundational definitions in place, we can define a component request:

Definition 5 (Component Request)
A component request $(\tau, \chi) \in T \times X$ represents a need for a component of type τ in context χ.

Injection typically begins with a component request in the empty context: $(\tau_0, \langle\rangle)$.

The policy by which the injector resolves a component request into a constructor graph is represented by a binding function:

Definition 6 (Binding Function)
A binding function $\mathcal{B} : T \times X \hookrightarrow C$ is a partial function such that for each (τ, χ) where $\mathcal{B}(\tau, \chi)$ is defined, $\mathcal{B}(\tau, \chi) \succ \tau$.

To implement the dependency resolution task of a dependency injector, therefore, we can use a binding function $\mathcal{B}$ to resolve the (recursive) dependencies of an initial type request $(\tau, \langle\rangle)$. This process yields a constructor graph G that can be used to directly instantiate the required components. The details of computing a constructor graph from a binding and a type request are the subject of the next section.

4.2 Resolving Component Requests

A component request is satisfied by a constructor graph that produces a component of the desired type, with all of its dependencies, consistent with the specified binding function. As in our implementation (section 5.2), we accomplish this with a two-step process:

1. Resolve the dependencies into a constructor tree.
2. Simplify the tree by merging common subgraphs.

The first step is implemented by algorithm 1, with the caveat that circular dependencies result in an infinite loop.

The second step produces a graph where possible component reuse is encoded in the graph itself. It is possible to reuse component implementations while instantiating directly from the tree by memoizing the instantiation process, but explicitly merging nodes that are candidates for reuse allows the graph to represent all possible component sharing while not precluding the instantiator from creating multiple instances if policy so dictates. Vertices can even carry additional attributes specifying such policy decisions.

More specifically, the aim of the second step is to produce constructor graphs with *maximal reuse*: each constructor appears on exactly one vertex for each unique transitive dependency configuration to which it applies. We would also like to detect and fail in the face of cyclic dependencies.

If the binding function is context-free ($\forall \tau, \chi . \mathcal{B}(\tau, \chi) = \mathcal{B}(\tau, \langle\rangle)$) — that is, the binding function always returns the same constructor for a particular type, irrespective

Algorithm 1 Resolving component request dependencies.

```
 1: function RESOLVE-DEPS($\tau, \chi, \mathcal{B}$)
 2:     $G \leftarrow$ new empty graph
 3:     $c \leftarrow \mathcal{B}(\tau, \chi)$
 4:     if $c$ is undefined then
 5:         fail there is no binding for $\tau$ in $\chi$
 6:     $v \leftarrow$ new vertex in $G$
 7:     $\mathrm{C}[v] \leftarrow c$
 8:     for $\tau' \in \mathcal{D}(c)$ do
 9:         $G' = (V', E') \leftarrow$ RESOLVE-DEPS($\tau', \chi +\!\!+ \langle c \rangle, \mathcal{B}$)
10:         $V \leftarrow V \cup V'$
11:         $E \leftarrow E \cup E' \cup \{v \xrightarrow{\tau'} \textsc{Root}[G']\}$
12:     return $G$
```

of the context in which it appears), then Resolve-Deps can be adapted into a depth-first graph traversal to address both problems.

If $\mathcal{B}$ is not context-free, the situation is more complicated. The arguments passed to a constructor may vary based on the context in which that constructor is used, and the same constructor may appear in multiple places in the constructor graph with different subgraphs for its dependencies. We say that such a constructor's dependencies are *divergent*. A constructor may have divergent dependencies even if it has the same bindings for all its direct dependencies due to divergence in its transitive dependencies, as in fig. 6. This means that it is difficult to determine whether a constructor's dependencies will be divergent. Further, it is technically possible to have several repetitions of a constructor before breaking a cycle. For example, consider the following:

$$c \succ \tau \qquad\qquad c' \succ \tau$$
$$\mathcal{D}(c) = \{\tau\} \qquad\qquad \mathcal{D}(c') = \{\}$$

$$\mathcal{B}(\tau, \chi) = \begin{cases} c' & \text{if } \chi = \langle c, c, c \rangle \\ c & \text{otherwise} \end{cases}$$

This configuration resolves to the constructor graph $c \xrightarrow{\tau} c \xrightarrow{\tau} c \xrightarrow{\tau} c'$. However, when resolving τ to c the second and third time, the resolution algorithm does not know whether the cycle will terminate. While this particular example is degenerate, if c has additional dependencies, such stacked configurations are not difficult to envision.

In practice, we solve the cyclical dependency problem by applying a depth limit to the constructor graph. It is unusual in our experience to have extremely deep object graphs, and the limit can always be increased if it is insufficient for a particular application, so this approach allows us to have a straightforward algorithm with an arbitrary but tunable limit instead of categorically rejecting entire classes of configuration structures.

Algorithm 2 is an efficient dynamic programming algorithm to perform Step 2, simplifying a tree into a graph with maximal reuse; this is the algorithm used by Grapht's implementation to merge a computed dependency tree with the valid dependency graph (section 5.2). The resulting graph has a single vertex for each

Algorithm 2 Simplify Constructor Graph

```
1: function SIMPLIFY-GRAPH($G$)
2:     $\langle v_1, \ldots, v_n \rangle \leftarrow$ TOPO-SORT($G$)[a]
3:     $G' = (V', E') \leftarrow$ new empty graph
4:     $\langle v'_1, \ldots, v'_n \rangle \leftarrow$ new list
5:     $m \leftarrow$ new map $C \times \mathcal{P}(V') \rightarrow [1, n]$
6:                          ▷ map constructors and dependency vertices to list positions
7:     for $i \leftarrow 1$ to $n$ do
8:         $D_i \leftarrow \{v'_j : (v_i, v_j) \in E\}$       ▷ $\forall v'_j \in D_i, j < i$ and thus already merged
9:         $c_i \leftarrow \mathrm{C}[v_i]$
10:        if $m(c_i, D_i)$ is defined then                  ▷ reuse previous vertex
11:            $j \leftarrow m(c_i, D_i)$
12:            $v'_i \leftarrow v'_j$
13:        else                                 ▷ constructor $\times$ deps unseen, add new vertex
14:            $v'_i \leftarrow v_i$
15:            add $v'_i$ to $V'$
16:            for $v'_j \in D_i$ do
17:                add $(v'_i, v'_j)$ to $E'$
18:            $m(c_i, D_i) \leftarrow i$
19:    ROOT[$G'$] $\leftarrow v'_n$
20:    return $G'$
```

[a] TOPO-SORT topologically sorts a DAG $G = (V, E)$ so that edges go from right to left ($(v_i, v_j) \in E \implies j < i$) and $v_n = \mathrm{ROOT}[G]$.

combination of a constructor and its transitive dependencies in the solution, but may use the same constructor for multiple vertices, thus achieving maximal reuse without sacrificing any generality. SIMPLIFY-GRAPH starts at the leaves of the tree and merges all possible vertices prior to merging the constructors that may depend on them.

That SIMPLIFY-GRAPH produces graphs with maximal reuse can be shown with strong induction:

Base case (v_1)

C[v_1] will have no dependencies; otherwise, v_1 would have outgoing edges and would not be the first node in the topological sort. Also, the subgraph rooted at v_1 has maximal reuse among the vertices seen so far: the algorithm has not encountered any other vertices, so there are no other vertices with which it might be redundant. Therefore, adding it to the currently-empty G' means G' has maximal reuse.

Inductive step (v_i, $i > 1$)

$G' = (V', E')$ has maximal reuse. Each vertex v_j for $j < i$ has a corresponding vertex $v'_j \in V'$ that is the root of a non-redundant subgraph of G'. The algorithm maps the dependencies of v_i in G to their corresponding vertices in G' (line 8), and looks up the constructor and this resolved dependency set in the table of nodes seen so far. If the constructor has already been applied to equivalent dependencies, then it will appear in the lookup table, and its resulting vertex (and subgraph) will be used, maintaining maximal reuse. If (c_i, D_i) does not appear in m, then either c_i has never been seen, or at least one vertex has a different configuration (otherwise, it would be in m). A new vertex is generated for this unique configuration and maximal reuse is preserved.

Since the singleton graph trivially has maximal reuse, and no subsequent iteration breaks the maximal reuse property, the final graph G' will have maximal reuse.

4.3 Bindings

So far, we have treated the binding function $\mathcal{B}(\tau, \chi)$ as a black box: given a type and a context, it returns a constructor if one is configured. This keeps our theoretical model independent of any particular representation of policy.

As discussed in section 3.2 several DI tools represent policy in terms of individual *bindings*, each of which binds a type to either a constructor or another type. In this section, we present a mathematical notion of bindings that can be used to define the behavior of the binding function.

Definition 7 (Binding)
A binding $b = (\bar{\tau}, \tilde{\chi}) \mapsto t$ *specifies the implementation to use, where*

- $\bar{\tau} \in T$ *is a type to match.*
- $\tilde{\chi}$ *is a* context expression, *a regular expression over contexts.*
- t *is a* binding target, *either a type or a constructor. The type resulting from t is denoted $\tau(t)$; $\tau(t) = t$ if t is a type, and $\tau(t) = \tau'$ if t is a constructor $c \succ \tau'$.*

To evaluate $\mathcal{B}(\tau, \chi)$, the resolver locates the matching binding $(\bar{\tau}, \tilde{\chi}) \mapsto t$ where $\bar{\tau} = \tau$ and $\tilde{\chi}$ matches χ. If t is a constructor, then $\mathcal{B}(\tau, \chi)$; if t is a type, then the bindings are evaluated recursively as $\mathcal{B}(t, \chi)$.

Within context expressions, any reasonable predicate over constructors can serve as the basis for atoms (defining the set of constructors that will be matched at a particular position). A natural choice is to use types, with a type τ matching any constructor $c \succ \tau$. If the runtime environment has a root type (the supertype of all types, such as Java's Object), that can serve as the wildcard.

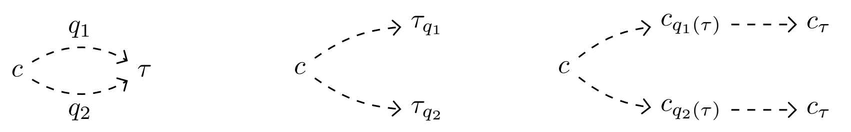
If there are multiple bindings that match (τ, χ), then the *most specific* is selected. The exact definition of specificity is dependent on the details of the runtime environment and type system; section 5.1 describes the specificity notion used in Grapht.

The rest of our model and algorithms are independent of the binding function representation, so alternate representations can be used without affecting the remainder of section 4.

4.4 Qualifiers

In section 3.3, we identified qualified dependencies as a useful feature for a DI container. Our model can be extended to allow dependencies to have qualifiers; further, qualifiers can be reduced to context-aware policy, so while they are convenient they can be seen as a sugar on top of a core model of dependency injection. This result combines with the examples requiring context-aware policy in section 3.4 to show that context-sensitive policy is strictly more expressive than qualified dependency matching.

As discussed in section 3.3, qualifiers are a convenient means of distinguishing between dependencies of the same type, particularly when a single component has



(a) Original (b) Transformed dependencies (c) Solution (unsimplified)

Figure 10. Reduction of qualified graph.

multiple such dependencies They are also useful for categorizing dependencies in dynamically-typed environments such as Python.

To add qualifiers to our model, we make the following amendments:

- Augment the runtime environment with a set of qualifiers Q. Qualifiers can be considered to be opaque labels. We designate a particular qualifier $q_\perp$ to represent the lack of a qualifier.
- Add qualifiers to constructor dependencies: $\mathcal{D} : C \to Q \times T$
- Add qualifiers to component requests, so they are expressed as triples (q, τ, χ).
- Add qualifiers to the signature of the binding function: $\mathcal{B}(q, \tau, \chi)$. Using a binding-based representation, each binding will be a 3-tuple $(\bar{q}, \bar{\tau}, \tilde{\chi})$, where $\bar{q}$ is a qualifier matcher that determines whether the binding applies to a particular qualified dependency. A designated qualifier matcher $\top$ matches any qualifier.
- Add qualifiers to constructor graph edge labels, so each edge is labeled with $(q, \tau) \in Q \times T$.
- Make contexts sequences of qualified constructors $\chi = \langle (q_1, c_1), \ldots, (q_n, c_n) \rangle$, where q_i is the qualifier on the edge leading to the vertex labeled with c_i.

The resolution algorithms pass the qualifier associated with each dependency to the binding function and associate the qualifiers with the correct edge labels.

However, it is not necessary to propagate the notion of qualifiers throughout the model. A runtime environment, binding function, and component request that use qualifiers can be reduced to ones that only make use of context-sensitive policies. Figure 10 is a graphical depiction of the process of reducing a constructor c with two qualified dependencies on τ. There are three steps to computing this reduction:

1. Replace each qualified dependency (q, τ) with a dependency on a synthesized type $\tau_q \subseteq \tau$. These synthetic types may be realized as actual types in the runtime environment, or they may exist only as bookkeeping entities in the DI container.
2. Modify the initial component request (q, τ, χ) to be $(q_\perp, \tau_q, \chi)$ if $q \neq q_\perp$.
3. Modify the bindings as follows:
   - Bind each synthetic qualifier type τ_q to a constructor $c_{q(\tau)} \succ \tau_q$ with $\mathcal{D}(c_{q(\tau)}) = \langle \tau \rangle$.
   - For each binding $b = (\bar{q}, \bar{\tau}, \tilde{\chi}) \to t$, substitute the binding $(\bar{\tau}, \tilde{\chi} + \langle \tilde{q} \rangle) \to t$, where $\tilde{q}$ matches any synthetic constructor $c_{q(t)}$ whose corresponding qualifier is matched by $\bar{q}$.

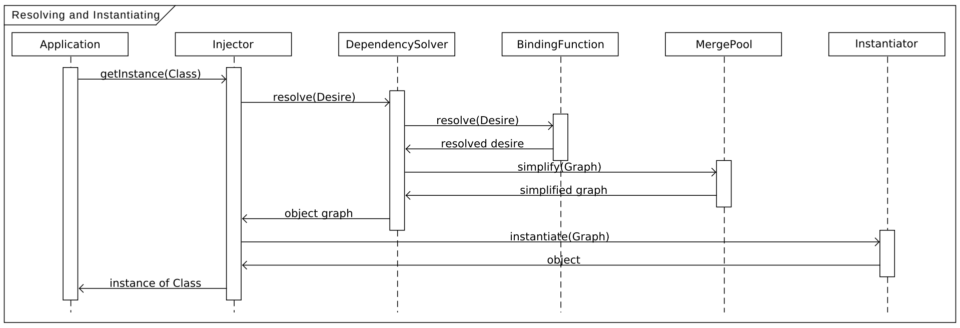


Figure 11. Simplified instantiation sequence diagram.

The reduction works by replacing each qualified dependency (q, τ) resulting in a constructor c with a constructor chain $c_{q(t)} \xrightarrow{\tau} c$. Any policy that examines the qualifier attached with a dependency type can instead look at the context to see if the type is being configured to satisfy the dependency of a synthetic constructor. While we have shown how to this for bindings, any computable binding function should admit a similar modification to look for contexts terminating in t_q.

After this reduction, the only qualifier in use is $q_\perp$ and the only qualifier matcher is $\top$, so qualifiers can be removed entirely.

5 Grapht

Grapht is our open-source dependency injector for Java built on the model in section 4. It is compatible with JSR 330 and passes its technology compatibility kit (TCK), the suite of tests provided to check JSR 330 compatibility; it can therefore be used as a replacement for existing containers in many situations. The code base is less than 6000 lines of Java, excluding tests. Its multi-phase design and graph-based approach have enabled us to build a simple, clean implementation that provides the novel features we require as well as common features generally expected of dependency injectors. Many of Grapht's public APIs are modeled after those of Guice, since it is well-known by Java developers.

Constructor graphs (definition 2) are the centerpiece of the design of Grapht. Constructors are represented in Grapht as Satisfaction objects; a satisfaction abstracts the underlying Java mechanisms for creating a class and determining its dependencies. Grapht provides different satisfaction implementations to instantiate classes, invoke provider objects, and return pre-constructed instances, allowing various types of bindings to be realized in a consistent fashion. We chose this term because Constructor is already the name of a core Java class that our code needed to use, and a constructor *satisfies* a request or desire for a component.

Grapht supports qualifiers; they are represented by Java annotations, consistent with JSR 330. Satisfaction implementations that instantiate component or provider classes determine the dependencies of those components by examining their constructors and methods for the annotations defined in JSR 330; most notably, the @Inject annotation marks a constructor or method as participating in the dependency injection setup process.

```
InjectorBuilder bld = new InjectorBuilder();
bld.bind(I1.class).to(B.class);
bld.within(Left.class, B.class)
   .bind(I2.class).to(C.class);
bld.within(Right.class, B.class)
   .bind(I2.class).to(D.class);

Injector inj = bld.build();
I1 obj = inj.getInstance(I1.class);
assert obj instanceof B;
```

Listing 2. Example code to build and use an injector.

```
bind I1 to B
within (Left, B) {
    bind I2 to C
}
within (Right, B) {
    bind I2 to D
}
```

Listing 3. Groovy injector configuration.

Figure 11 shows the high-level components that are involved in resolving and instantiating an object. The application (far left) requests an instance of a component type (class or interface), and the remaining components (all part of Grapht) work together to do the following:

1. Identify the appropriate implementation from application classes and policy.
2. Recursively resolve the component's dependencies.
3. Simplify the graph to identify shared components.
4. Instantiate the objects.

In this section, we will focus on two aspects of Grapht: how component policy is represented and resolved, and how Grapht builds and instantiates component graphs.

5.1 Dependency Configuration

Listing 2 demonstrates how to construct an injector and configure it with explicit bind rules using Grapht's Java API. InjectorBuider is a builder that generates additional bind rules for intermediate and super types and Injector performs the injection and instantiation of requested types. The `within` and `at` methods control the context sensitivity of the binding. There are `bind` variants that accept qualifiers to configure a qualified type binding rule. LensKit augments this Java API with a Groovy-based embedded domain-specific language that allows a more structured configuration style as shown in listing 3.

Grapht represents the binding function embodying the application's configured dependency policy as a series of individual binding functions, several of which are represented as sets of bindings. A typical GRAPHT configuration consists of the following binding functions, in decreasing order of precedence, to implement the subtyping behavior described in section 3.5 and allow defaults to supplement application-specified configuration:

1. *Explicit bind rules* — bindings for $T \mapsto \mathsf{C}$, that match type T exactly. T may be qualified and/or be active in limited contexts.

2. *Intermediate bind rules* — bindings generated from an explicit $T \mapsto \mathsf{C}$ rule: $U \mapsto \mathsf{C}$ for every type U that is both a subtype (exclusive) of T and a supertype (inclusive) of C.

3. *Supertype bind rules* — bindings generated from an explicit $T \mapsto \mathsf{C}$ rule: $S \mapsto \mathsf{C}$ for every supertype (inclusive) of T.

4. *Provider bindings* — bindings that match a request for a provider component of type Provider<T> and satisfy it with a provider that depends on and returns T. These bindings are primarily to support circular dependencies and are discussed in more detail in appendix A.1.

5. *Defaults* — default bindings declared within the Java source code (this binding function is not implemented as a set of bindings, but rather looks up defaults using Java reflection).

To handle subtypes, GRAPHT generates additional bind rules for each explicit binding. Given a binding between an interface and implementation, bindings are automatically generated for the intermediate types between the interface and implementation types, and supertypes of the interface. This cleanly allows dependencies on a sub- or supertype of the bound interface to receive the implementation while letting it be overridden with a new explicit rule. We prefer this approach over integrating subtype matching logic into the individual bind rules because it allows rule matching to be independent of the right-hand side of the rule.

GRAPHT implements qualifiers directly, rather than reducing them to context with synthetic constructors. This is partly for implementation simplicity, and partly for historical reasons. This decision may be revisited in a future version of the software.

GRAPHT compares bindings for specificity by first looking at the qualifier matcher. Since only exact types match, not subtypes, we do not need to consider the component's type to determine specificity. A qualifier match that matches the exact annotation value is the most specific, followed by matchers that select qualifiers by type (the most common type of qualifier matcher), and lastly by the universal qualifier matcher. If multiple bindings with the same qualifier matcher match the dependency, the binding function considers the specificity of their context matches. A context match a is more specific than b if it has a non-wildcard match closer to the end (instantiated type) than b; if the last non-wildcard match is the at the same position, then the positions of the previous non-wildcard matches are considered, preferring matches that match more elements; finally, if two matches have equal length and position, the tie is broken by considering the specificity of the qualifier and type match. This allows the programmer to specify policy by matching some type deep in the graph and know that their new binding will take precedence over bindings further out in the graph.

If two matching bindings in a single binding function are equivalent — they have qualifier matchers of equal specificity, and their context matches are equal — then GRAPHT fails with an error indicating that the bindings are ambiguous. Applications should not specify multiple applicable bindings without using multiple binding functions to indicate a clear precedence relationship. There may be a way to statically analyze a set of bindings to determine possible ambiguity; we would like to explore this in the future.

These rules are, admittedly, somewhat arbitrary. We have strived to develop rules that will allow for useful configurations with minimal surprise to the developer. Space does not permit us to provide a full justification for each rule in this context.

5.2 Graph Building and Instantiation

Grapht maintains a valid dependency graph for all requested types as the application executes. As additional components are requested by the application, Grapht updates this graph to have their types and to reuse components instantiated by previous requests when appropriate. The dependency graph is computed and validated before any requested components are instantiated. The dependency graph is also exposed to the application, so that advanced applications can process and modify the graph in useful, application-specific ways.

The standard implementation of Injector maintains a synthetic root node that has edges to all components that have been requested. When the application requests a component from the injector, and that component has not yet been resolved, the injector does the following:

1. Build a dependency tree for the requested component.
2. Add a new edge from the synthetic root to the root of the dependency tree.
3. Simplify the resulting graph with algorithm 2, augmenting the topological sort to require that nodes in the new component's dependency tree come after nodes in the pre-existing component graph. This ensures that the new component and its dependencies can re-use existing components wherever possible.

Once a graph for a component is available, instantiation proceeds by recursing through the graph and invoking the constructors stored at each node. Instantiation is quite simple because Grapht maintains valid dependency graphs at all times: if Grapht is able to produce a graph, then the only remaining errors that can occur are internal object instantiation errors. Individual graph nodes can have a cache policy to determine whether their instances are memoized or reinstantiated for each request. The constructor graph represents all possible component reuse as shared nodes, and defers actual object sharing decisions to instantiation time.

6 Effectiveness of Grapht

Having laid out our model of dependency injection, and discussed how it is used to realize an effective DI tool for Java, we now turn to the particular functionality that our approach enables. In this section, we describe several features, primarily of the LensKit toolkit, that are enabled by Grapht's graph-based approach, and the surprising usefulness of context-sensitive policy.

6.1 Identifying Sharable Components

As discussed in section 3.6, it is useful in certain applications to have components that live in different scopes. This is a common requirement in web applications and software that integrates with them. The core issues that affect the ability for components to be shared across web requests are thread safety and database access. In many common architectures, web applications open database connections or obtain them from a pool on a per-request basis. Any component that needs access to the database therefore needs to be instantiated for each request, so it can access the database connection in use for that request.

In LensKit, recommender model components can typically be designed to be both immutable and independent of the database once they have been computed. These components can be instantiated once and shared across requests; then can also be pre-instantiated and serialized by a model-builder process and loaded by the web application server, possibly residing on a separate machine. It can also take many hours to compute some of the statistical models encapsulated in these components. The process of building a model may require access to the database at construction time (e.g. to obtain users' purchase histories) but, once constructed, the model is independent of the database.

Computing the object graph as a first-class entity prior to object instantiation allows LensKit, guided by some light annotations, to automatically identify objects that can be pre-computed and/or shared between different uses of the recommender. This same logic can also be used to enable automatic separation or validation of component scoping in general web applications.

Without Grapht's static analysis and graph rewriting support, algorithm implementers or application developers would need to manually determine which components are sharable, write code to instantiate them, and integrate those instantiated objects with their final configuration.

6.1.1 Building a Model

The LensKit model build process does the following:

1. Using Grapht, build a graph from the algorithm DI configurations.
2. Traverse the graph, looking for sharable components. Each sharable component is instantiated immediately, and its node is replaced as if it were the result of an instance binding.
3. Encapsulate the resulting graph, with sharable components pre-instantiated, in a *recommender engine*.
4. For each recommendation request, create a LenskitRecommender containing a copy of the instantiated graph (sharing the instances of sharable components) and unique instances of non-sharable components.

In order to support these manipulations, LensKit introduces two annotations to indicate the way components should be handled by the recommender engine builder. One, @Shareable, marks a component as a candidate for sharing. By applying the shareable annotation, the developer warrants that the component is thread-safe and usable across requests. It must be applied to the implementation, not to the interface; the same interface may well have both shareable and non-shareable implementations.

The second, `@Transient`, applies to dependencies of a component and indicates that the labeled dependency is only needed during construction. It constitutes a promise that, once the constructor or provider has built an object, the object is free of references to the dependency. This allows shareable components to access non-shareable components (such as a data access object) during their construction phases, so long as they do not retain references to them.

In step (2) above, LensKit scans the constructor graph to identify all shareable components that have no non-transient dependencies on non-shareable components. It then instantiates them and builds a replacement graph. The resulting constructor graph is equivalent to the result of manually pre-instantiating each shared component and using these instances in a configuration. This allows algorithm authors to leverage a lot of framework assistance for deploying their algorithms in realistic environments with just a few Java annotations.

The modified graphs, with the pre-instantiated shared components, can also be serialized to disk and reloaded later. We use this to compute recommender models in a separate process from the web server; once a new model is computed and saved, the web server notices and reloads its version of the model from disk.

6.1.2 Sharing in Evaluation

Recommender developers often need to try multiple variants of an algorithm on a data set to determine the best configuration for their application. This can be an expensive process, as the recommender's models must be computed for each variant. But not all variations affect the model class: some variations may be in the components that use the models.

LensKit provides a tool for running multiple algorithms and variants on a data set. It takes advantage of Grapht's static analysis support to reuse common components between configurations, so that a particular model configuration only needs to be computed once for all variations that use it. The LensKit evaluator processes all the algorithms configured for a evaluation and uses Grapht to create their component graphs before training or running any of them. It then combines all graphs using the same simplification mechanism as Grapht's default injector, so any component configurations shared by multiple components are represented by common subgraphs. Reusing computed models across compatible variants is then simply a matter of caching the result of instantiating each graph node.

The evaluator also establishes dependency relationships between individual evaluation jobs (evaluating a single algorithm on a single data set) so that a single use of a common component is built first, with other uses waiting for it to complete. This allow a multithreaded evaluation to start working on other algorithms that do not share common components instead of starting 8 evaluations that will all block on the same model build. We are exploring more fine-grained approaches to the parallel model building process using the fork-join framework introduced in Java 7, and being able to build a single graph of all components in a large experiment will be very helpful in this endeavor.

6.2 Visualization and Diagnostics

Pre-computing dependency solutions also provides benefits for debugging recommender configurations and applications. First, it ensures that dependency problems fail fast. Since recommendation models may be costly to compute and web servers slow to

```
within (UserSimilarity) {
    bind VectorSimilarity to SpearmanCorrelation
}
within (ItemSimilarity) {
    bind VectorSimilarity to CosineVectorSimilarity
}
```

Listing 4. Configuring user and item similarity functions for LensKit.

launch, it is useful to fail quickly rather than have dependencies of some component fail only after spending hours computing a model or waiting until an unlucky user accesses a faulty request. The multi-stage approach employed by Grapht ensures that all dependency errors are identified as early as possible, as they will result in a failure to build a constructor graph instead of a failure to instantiate needed objects. Constructing the graph can even be added to the build validation if the build system is sufficiently powerful, allowing dependency errors to be detected at application compile time instead of manifesting later as runtime errors.

Statically analyzable graphs also allow us to provide diagnostic tools such as automatic diagramming of configurations without incurring the cost of object instantiation. Guice and PicoContainer provide support for inspecting and diagramming configurations, but they both accomplish this by instrumenting the instantiation process. Grapht allows the dependency solution, representing the final object graph, to be computed independently of object instantiation, allowing for straightforward tooling support without requiring expensive computation or access to suitable input data.

6.3 Utility of Contextual Policy

The availability of context-aware policy in Grapht has influenced the design of several LensKit components and reduced the need for redundant qualifiers. One case where this applies is in the similarity components used by the nearest-neighbor collaborative filters that need to be able to measure the similarity between users or items in the system. We define specific types for comparing users and items (*UserSimilarity* and *ItemSimilarity*, respectively), but many similarity functions, such as cosine similarity, just operate on a vector and do not care if it is for a user or an item. We therefore provide a generic *VectorSimilarity* class that implements vector similarities without item or user IDs, and provide default implementations of the user- and item-specific similarity functions that delegate to a vector similarity. If we only had qualifiers, not context-aware policy, to control implementation decisions, we would annotate the vector similarity with a @User or @Item annotation to allow different vector functions to be used for comparing users or items. However, this information is redundant with the fact that the vector similarity is required by the user or item similarity class, and provides no convenience over using a context-sensitive binding when context-aware policy is easy to use. For example, the configuration in listing 4 will result in using Spearman for comparing users and cosine similarity for items.

This design has several benefits:

- Fewer annotation types must be maintained.
- More information is encoded directly into the classes themselves.

```
within (Left, ItemScorer) {
    within (ItemSimilarity) {
        bind VectorSimilarity to PearsonCorrelation
    }
}
within (Right, ItemScorer)
    within (ItemSimilarity) {
        bind VectorSimilarity to SpearmanCorrelation
    }
}
```

Listing 5. Configuring a hybrid item scorer.

- Documentation is simplified and easier to read, particularly as a result of encoding the information about how a class is used into the types.

Moving beyond slightly different syntax, context-aware policy also allows us to create configurations we could not create with only qualifiers. For example, we may want to configure different item similarities for different components of a single configuration; this is a specific instance of the problem in fig. 6. With context-aware policy, we can just specify enough information to find the location where we want each similarity function. For example, listing 5 will configure a hybrid of two differently-configured item-item recommenders.

Context-aware policy allows us to achieve these kinds of results and compose individual components into arbitrarily-complex configurations without implementing verbose, error-prone custom wrapper components to expose the particular configuration points needed as qualifiers. Also, with context-aware policy available, many qualifiers (such as qualifying the *VectorSimilarity* dependencies to indicate whether they apply to users or items) become redundant and add no clarity to the design or configuration.

7 Conclusion

We have described a new dependency injection toolkit for the Java platform that allows a greater degree of composability for software components and better support for static analysis of object graphs than existing solutions provide. These capabilities have enabled several time- and effort-saving features in the LensKit recommender systems toolkit. We have also described a mathematical model of dependency injection that abstracts Grapht's capabilities and provides a basis for formal treatment and, we hope, further research on dependency injection.

Our approach to context-aware policy allows expressive matching on deep contexts with easy configuration. For configuring recommender applications, this allows LensKit's individual components to be reconfigured into arbitrarily complex configurations, allowing extensive code reuse. We hope that the improvements to component composability and code reuse will prove useful in a host of other applications as well. We have also shown that context-aware policy is strictly more powerful than the dependency qualifiers provided by many current dependency injection frameworks; while we expect that qualifiers will live on due to their convenience, they can be viewed as a syntax sugar on top of a more expressive paradigm.

There are a variety of extensions to dependency injection that may be worth considering in the future. One is *weighted dependency injection*: under this scheme, constructors or bindings have associated weights expressing the cost of using them, and the injector tries to find the lowest-cost solution to the component request. This problem is likely NP-hard.

Opportunistic dependency injection is a simplified extension that is likely more practical. In opportunistic DI, some optional dependencies are marked as "opportunistic", meaning that they will only be instantiated and used if required by some other component as a non-opportunistic dependency. They differ from normal optional dependencies in that an optional dependency will be supplied if it is possible to satisfy the dependency given the binding function, while an opportunistic dependency is only supplied if the configured constructor will be invoked to satisfy some other non-opportunistic dependency in the final object graph. The key use case for this extension is when a component A can operate more efficiently if an expensive component B is available, but the efficiency gain alone is not sufficient to warrant the cost of instantiating B. If some other component requires B, however, then A can take advantage of it under opportunistic DI.

Grapht has proven to be a valuable tool in making LensKit flexible and easy to use, and we have found it suitable for use in more traditional dependency injection applications (i.e. web applications) as well. We hope that its well-defined model and straightforward implementation will make it a useful platform for future developments in dependency injection.

References

[CI05] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *ECOOP 2005 — Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer Berlin Heidelberg, 2005. `doi:10.1007/11531142_6`.

[Eks14] Michael D. Ekstrand. *Towards Recommender Engineering: Tools and Experiments in Recommender Differences.* Ph.D Thesis, University of Minnesota, Minneapolis, MN, July 2014. URL: `http://hdl.handle.net/11299/165307`.

[ELKR11] Michael D. Ekstrand, Michael Ludwig, Joseph A. Konstan, and John T. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and LensKit. In *Proceedings of the fifth ACM conference on Recommender systems*, RecSys '11, pages 133–140, New York, NY, USA, 2011. ACM. `doi:10.1145/2043932.2043958`.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, January 2004. URL: `http://martinfowler.com/articles/injection.html`.

[Ham11] Paul Hammant. PicoContainer, November 2011. URL: `http://picocontainer.org`.

[HH13] Shrinidhi R. Hudli and Raghu V. Hudli. A Verification Strategy for Dependency Injection. *Lecture Notes on Software Engineering*, 1(1):71, 2013. `doi:10.7763/LNSE.2013.V1.16`.

[JL09] Rob Johnson and Bob Lee. JSR 330: Dependency Injection for Java. Technical Report 330, Java Community Process, October 2009. URL: `https://jcp.org/en/jsr/detail?id=330`.

[Joh02] Rob Johnson. Spring Framework, 2002. URL: `http://www.springsource.org/spring-framework`.

[Koh12] Nate Kohari. Ninject, 2012. URL: `http://www.ninject.org/`.

[LBW09] Bob Lee, Kevin Bourrillion, and Jesse Wilson. Google Guice: A lightweight dependency injection framework for Java 5 and above, 2009. URL: `http://code.google.com/p/google-guice`.

[Mar96] Robert C. Martin. The Dependency Inversion Principle. *C++ Report*, 8(6), May 1996.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *Software Engineering — ESEC '95*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer Berlin Heidelberg, 1995. `doi:10.1007/3-540-60406-5_12`.

[Pra09] Dhanji R Prasanna. *Dependency Injection*. Manning, Greenwich, Conn., 2009. URL: `http://www.manning.com/prasanna/`.

[PSC$^+$10] Erick B. Passos, Jonhnny Weslley S. Sousa, Esteban Walter Gonzales Clua, Anselmo Montenegro, and Leonardo Murta. Smart composition of game objects using dependency injection. *Computers in Entertainment*, 7(4):53:1–53:15, January 2010. `doi:10.1145/1658866.1658872`.

[RJ07] Ekaterina Razina and David Janzen. Effects of Dependency Injection on Maintainability. *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, pages 7–12, November 2007. URL: `http://digitalcommons.calpoly.edu/csse_fac/34`.

[Wil12] Jesse Wilson. Dagger, 2012. URL: `https://square.github.io/dagger/`.

[YTM08] Hong Yul Yang, E. Tempero, and H. Melton. An Empirical Study into Use of Dependency Injection in Java. In *19th Australian Conference on Software Engineering, 2008. ASWEC 2008*, pages 239–247. IEEE, March 2008. `doi:10.1109/ASWEC.2008.4483212`.

About the authors

Michael D. Ekstrand is an assistant professor in the Department of Computer Science at Texas State University. He is also the lead developer of LensKit, an open-source toolkit for supporting reproducible research on recommender systems.

He was previously with GroupLens Research at the University of Minnesota, where the bulk of this work was conducted.

Contact him at `ekstrand@acm.org`, and more information on his research activities is available from `http://md.ekstrandom.net/research/`.

Michael Ludwig is a Ph.D candidate in computer graphics at the University of Minnesota. Prior to entering the Ph.D program, he worked with the GroupLens Research group, where he made significant contributions to the design of Grapht and wrote its initial implementation. His work currently focuses on realtime rendering and its applications to human perception.

Contact him at `mludwig@cs.umn.edu`.

Acknowledgments This work would not have been possible without the support and assistance of current and former colleagues in GroupLens Research. We are particularly grateful to Rich Davies for suggesting dependency injection as a solution to our LensKit design problems; to Shilad Sen, Jennifer Ekstrand, Max Harper, and Joseph A. Konstan for providing valuable feedback on draft versions of this paper; and to John T. Riedl for supporting and supervising early portions of the work.

We also thank the anonymous reviewers for their thorough and helpful feedback that helped us to significantly improve the manuscript.

This work has been funded in part by the National Science Foundation under IIS grants 05-34939, 08-08692, 08-12148, and 10-17697.

A Circular Dependencies

Many DI containers support circular dependencies among components, and JSR 330 mandates this capability. Grapht has optional support for circular dependencies in order to comply with the JSR and pass its compatibility tests. In LensKit, we disable circular dependency support.

It is good to avoid circular dependencies for several reasons:

- Instantiating circular dependencies is awkward, as objects need to be able to be partially initialized and passed to each other before initialization is complete.
- Because objects can obtain references to partially-initialized objects during the instantiation process, there is greater opportunity to misuse objects (invoke methods on them before initialization is complete). In the absence of circular dependencies, a reference to an object is only made available to other components once the object is fully instantiated and initialized, so it is impossible for this particular type of runtime error to occur.
- Circular dependencies can often be factored out, resulting in more loosely-coupled class design. This can be done by either injecting both components into a third that mediates their interaction, or injecting a common component into each of the formerly mutually-dependent components. For example, a circular relationship between an observer and observee can be refactored by injecting a publish-subscribe event bus into both of them, or by injecting the observee into the observer and having the observer register itself with the observee.

In Java, and many other environments, circularly dependent components cannot be instantiated if their dependencies are only expressed via constructor parameters. The cycle must be broken either by having some dependencies injected into fields or setter methods, or by injecting providers of required components rather than the components themselves at some point in the cycle. In both cases, the components must allow for the circular dependency in their design, either by depending on a provider or by exposing dependencies via fields or setters.

A.1 Providers and Cyclic Dependencies

JSR 330 mandates supporting cyclic dependencies, which presents a challenge when first creating a dependency tree or working with a directed acyclic graph. One way of supporting cyclic dependencies is if types A and B have a cyclic dependency, A can instead depend on Provider<B>. Although this may break cycles when using DI containers that instantiate and resolve lazily, including a dependency to a provider of a type does not break the cycle in a constructor graph.

When Grapht encounters a dependency on a provider while building the initial solution tree (satisfied by a provider binding), it doesn't resolve the provider's dependencies immediately. It instead adds the provider to a queue of deferred components to be processed after all non-deferred dependencies have been resolved. Grapht then simplifies the graph of non-deferred components and begins processing the deferred components one by one, simplifying after each, until all dependencies are resolved. Using the simplification phase means that the provider component will depend on the same graph nodes as — and therefore be able to share objects with — other uses of the component, including uses that make the dependency cyclic.

Cyclic dependencies mean that the final object graph will contain cycles that must be represented in the instantiation plan. To reduce the impossible states exposed to code that analyzes dependency graphs and makes no use of circular dependencies, Grapht's data structures are built around rooted DAGs; when circular dependencies are supported, the dependency solver maintains a separate list of back edges completing dependency cycles. The instantiator consults the back edge table if it cannot find some required dependency in the graph itself. LensKit does not enable Grapht's cyclic dependency support, and can therefore ignore the back edge table (it will always be empty). This has the side effect of preventing LensKit components from depending on providers, but this has not been a problem.