Decouplink: Dynamic Links for Java

Jensen, Martin Lykke Rytter; Jørgensen, Bo Nørregaard

Published in:
Lecture Notes in Computer Science

DOI:
10.1007/978-3-642-23798-0_43

Publication date:
2011

Document version
Final published version

Citation for pulished version (APA):
Jensen, M. L. R., & Jørgensen, B. N. (2011). Decouplink: Dynamic Links for Java. In *Lecture Notes in Computer Science: Proceedings of the 5th European Conference on Software Architecture* (Vol. 6903, pp. 422 - 437). Springer. https://doi.org/10.1007/978-3-642-23798-0_43

Decouplink: Dynamic Links for Java

Martin Rytter and Bo Nørregaard Jørgensen

The Maersk Mc-Kinney Moller Institute, University of Southern Denmark,
Campusvej 55, 5230 Odense M, Denmark
`{mlrj,bnj}@mmmi.sdu.dk`
`http://www.sdu.dk/mmmi`

Abstract. Software entities should be open for extension, but closed to modification. Unfortunately, unanticipated requirements emerging during software evolution makes it difficult to always enforce this principle. This situation poses a dilemma that is particularly important when considering component-based systems: On the one hand, violating the open/closed principle by allowing for modification compromises independent extensibility. On the other hand, trying to enforce the open/closed principle by prohibiting modification precludes unanticipated dimensions of extension. Dynamic links increase the number of dimensions of extension that can be exploited without performing modification of existing types. Thus, dynamic links make it possible to enforce the open/closed principle in situations where it would otherwise not be possible. We present Decouplink – a library-based implementation of dynamic links for Java. We also present experience with the use of dynamic links during the evolution of a component-based control system.

Keywords: Dynamic links, extensibility, object-oriented programming.

1 Introduction

The inability to close software components to modification poses a threat to the extensibility of software systems [27]. Ideally, individual software components are open for extension, but closed to modification [17,15].

The need for modification arises when a software component does not comply with its specification – i.e. due to a bug – or when there is a need to incorporate new requirements. Whereas bugs rarely pose an enduring problem, the need to incorporate new requirements does. This is so, because all non-trivial software systems are subject to uncertainty, which requires them to evolve in ways that cannot be anticipated [12,13,3]. Thus, the need for modification to accommodate extension is usually an enduring problem.

Modifications that introduce new functionality are not only enduring, they also tend to be more difficult to confine. Whereas correcting a bug can often be confined so that dependent components remain unaffected, incorporating new functionality is more likely to affect existing components.

Software evolution implies that a software system must change in order to support new requirements. However, components inside the system do not per se

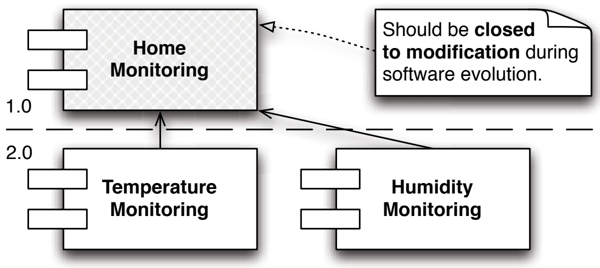

Home Monitoring
Should be **closed to modification** during software evolution.
1.0
2.0
Temperature Monitoring
Humidity Monitoring

Fig. 1. Evolution of a component-based home monitoring system

need to change. In the best case, new functionality can be introduced by adding new components, while existing components remain closed to modification.

We will use the simple home monitoring system in figure 1 to discuss the open/closed principle – in figure 6 we will share experience from the evolution of a real system. To start with, the home monitoring system consists of a single component, i.e. `home monitoring`. In its next version, two new components are added to the system, i.e. `temperature monitoring` and `humidity monitoring`.

To satisfy the open/closed principle, it must be possible to add the two new components in the system without modifying the `home monitoring` component. Unfortunately, it is not always possible to anticipate those dimensions of extension – i.e. "kinds of" extension – that will be needed in the future. When this is the case, extension developers are faced with an inconvenient dilemma:

One the one hand, an extension developer – e.g. the developer of `temperature monitoring` – may choose to violate the open/closed principle by performing modification of an existing component – e.g. `home monitoring` – to facilitate introduction of the extension. Even if the required modification seems to be backwards compatible, the fact that it is made by an extension developer makes it problematic. The problem is that other extensions may require similar modifications that could potentially produce a conflict. Therefore, the composition of invasive extensions must entail a global integrity check, and thus extensions cannot be completely independent – i.e. the system fails to be independently extensible [26].

On the other hand, an extension developer may refrain from any modification of existing components – i.e. `home monitoring` remains closed to modification. This decision implies that the required extension – e.g. `temperature monitoring` – cannot be introduced. Thus, the open/closed principle is violated as the system fails to be open for extension.

In summary, enforcement of the open/closed principle relies on anticipating required dimensions of extension. The ability to do so is one of the most important skills for a software architect to master. Nevertheless, even the most skilled software architect can never anticipate everything – thus, in the ultimate case the open/closed principle cannot be enforced.

An important feature of component platforms is the ability to handle situations, where modification of existing components is unavoidable. This is traditionally done by implementing a component lifecycle management system that maintains dependencies among component versions [16]. While versioning is certainly always an option, it should be the last option. In general, it is best if modification of existing components can be avoided.

In this paper, we argue that the need for modification of existing components can be reduced. It is often the case that existing components must be modified, not to change existing functionality in any fundamental way, but to allow new components to associate new functionality with concepts managed by existing components. We will demonstrate that this form of modification can be avoided – and thus our ability to satisfy the open/closed principle can be increased.

The core of our approach is dynamic links – a new kind of link that can connect objects of unrelated types. Dynamic links promote both elements of the open/closed principle: First, dynamic links promote openness by allowing new objects to be attached to old objects in ways that were not anticipated. Second, dynamic links can connect objects without modifying their types – existing types remain closed to modification.

The paper is a continuation of preliminary work described in [23]. It provides two main contributions: We present Decouplink [1] – a library-based implementation of dynamic links for Java [2] – and we present experience with the use of dynamic links to evolve a component-based control system for greenhouse climate control.

The paper is organized as follows. We introduce dynamic links in section 2. Section 3 presents support for dynamic links in Java. In section 4, we present experience with the use of dynamic links during the evolution of a greenhouse control system. Section 5 presents related work. Section 6 concludes the paper.

2 Dynamic Links

In this section we introduce dynamic links, we discuss how they are different from traditional links, and we demonstrate that dynamic links promote the open/closed principle.

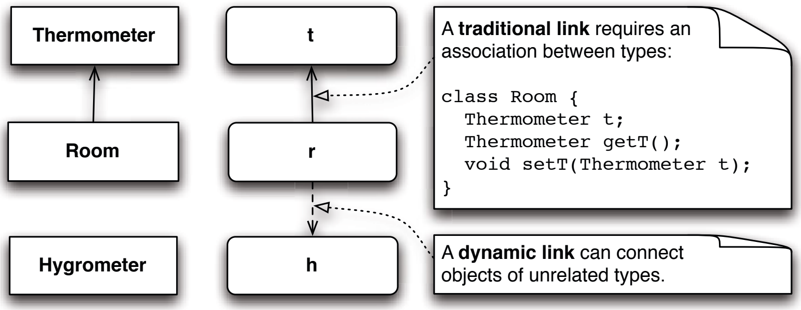
In object-oriented software, a *link* is a connection between two objects. A link usually has a direction and connects exactly two objects – a *source object* and a *destination object*. Links enable us to represent complex domain concepts as compositions of primitive objects connected by links. The use of a link between objects commonly relies on an *association* between types. The relationship between object-based links and type-based associations is emphasized in the UML specification [19]:

> "An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end."

[1] Get Decouplink from `http://decouplink.com`.

Given the definition above, a traditional link may be thought of as "an instance of" an association. It is only possible to create a link when a corresponding association exists. In Java, an association usually manifests itself as a field. E.g. it is only possible to connect a `Room` object and a `Thermometer` object when a suitable field has been declared, e.g. `Room.thermometer`.

A *dynamic link* can connect objects of unrelated types. Unlike a traditional link, a dynamic link does not rely on the declaration of an association. It is therefore possible to create a dynamic link between any two objects.


Thermometer
Room
Hygrometer
t
r
h
A **traditional link** requires an association between types:
A **dynamic link** can connect objects of unrelated types.

```
class Room {
  Thermometer t;
  Thermometer getT();
  void setT(Thermometer t);
}
```

Fig. 2. Comparison of traditional links and dynamic links

The difference between traditional links and dynamic links is illustrated in figure 2. The example shows three objects. The `r` object is an instance of the `Room` type – similarly, `t` is an instance of `Thermometer`, and `h` is an instance of `Hygrometer`. The figure shows two links:

First, `r` and `t` are connected by a traditional link. The link can exist only because a corresponding association exists between `Room` and `Thermometer`. In the code, the association is implemented using a field and two accessor methods.

Second, `r` and `h` is connected by a dynamic link. The dynamic link is drawn using a dashed line. The link is possible even though `Room` and `Hygrometer` do not participate in a type-based association. Thus, no methods or fields in the `Room` type depend on the `Hygrometer` type.

The primary advantage of dynamic links is that they promote closing existing code to modification. This is the case because they, unlike traditional links, can connect new objects of unanticipated types without modifying existing types.

Figure 3 illustrates how dynamic links promote closing components to modification in situations where traditional links do not. The three components are similar to those in figure 1, and the types and objects provided by each component are similar to those in figure 2.

First, the `temperature monitoring` component uses a traditional link to extend the system. The link connects `t`, an instance of the new type `Thermometer`, to `r`, an instance of `Room`. As we have previously seen, this link can only be created when there exists an association between `Room` and `Thermometer`. Thus, modification of the original `home monitoring` component is required.

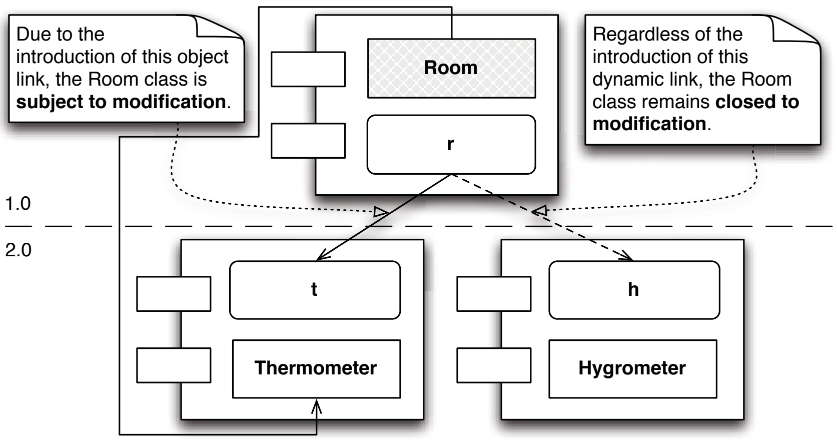

Due to the introduction of this object link, the Room class is **subject to modification**.
Room
r
Regardless of the introduction of this dynamic link, the Room class remains **closed to modification**.
1.0
2.0
t
Thermometer
h
Hygrometer

Fig. 3. Dynamic links promote closing components to modification

Second, the `humidity monitoring` component uses a dynamic link to extend the system. It connects `h`, a `Hygrometer`, to `r`, a `Room`. Since the new link requires no corresponding association, no modification of `Room` is required to perform the extension – the `home monitoring` component remains closed to modification.

The benefits and limitations of dynamic links can be emphasized by distinguishing two kinds of extension:

– *Unanticipated structural extension* is the ability to create links from original objects to new objects of unanticipated types – e.g. "add a hygrometer to a room". Unanticipated structural extension is supported by dynamic links.
– *Unanticipated behavioral extension* is the ability to wrap unanticipated behavior around original behavior – e.g. "when the light is turned on, also turn on the heat". Unanticipated behavioral extension is not supported by dynamic links.

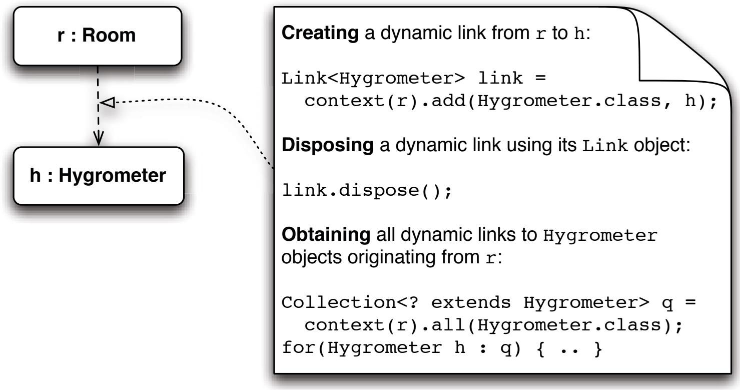
Unanticipated behavioral extension can only be achieved by modifying original types. This modification may be explicitly performed by the programmer – e.g. direct modification of source code. It may also be automated – e.g. load-time weaving of aspects [10]. Even automated modifications should be avoided to preserve independent extensibility [20].

The use of dynamic links is to some extent analogous to the way we "connect objects" in the physical world. The architect of a room is likely to create a room layout (a room type) without thinking about hygrometers (an associated type) – however, this does not prohibit a future owner of a room (an instance) from installing one. Similarly, type developers can never anticipate everything – should this prohibit object owners from creating links? As indicated above, we do not think so. However, we must stress not to use a comparison with the physical world to be an argument for or against dynamic links. We use the comparison merely to offer a familiar way of thinking about the role of dynamic links.

3 Design and Implementation

In this section we will present Decouplink – our implementation of dynamic links for Java. We show simple usage examples, we discuss the most notable design decisions, and we give an overview of how it works.

We have implemented Decoupling as a library. This choice makes the implementation accessible, as no language extension is needed.



Creating a dynamic link from r to h:

```
Link<Hygrometer> link =
  context(r).add(Hygrometer.class, h);
```

Disposing a dynamic link using its Link object:

```
link.dispose();
```

Obtaining all dynamic links to Hygrometer objects originating from r:

```
Collection<? extends Hygrometer> q =
  context(r).all(Hygrometer.class);
for(Hygrometer h : q) { .. }
```

Fig. 4. Simple usage of dynamic links

Figure 4 shows how to use our library to create, dispose, and obtain dynamic links. The `context()` method plays an important role. It is used to select an object on which to perform an operation, e.g. create a link or obtain existing links. The `context()` method is static and provided by a class in our library. By statically importing the method, it can be made available anywhere.

Since dynamic links do not rely on type-level associations, it is not possible to qualify links using accessor methods. Instead, we rely on *type-based link qualification*, i.e. we qualify links by the type of their destination object – not the name of a method or field. Consequently, instead of writing `r.addHygrometer(h)`, we write `context(r).add(Hygrometer.class, h)`. Type-based links qualification has two important consequences:

– It is always possible to add links to objects of new types without modifying existing types.
– The type of a destination object must be sufficiently specific to reveal the purpose of the data it represents. E.g. a person's first name should probably be of type `FirstName` and not merely `String`.

When programming an extension that adds new objects using dynamic links, it is often useful to be able to "protect" object links, so that other extensions cannot remove them. E.g. the `humidity monitoring` component should be able to ensure that no other extension intentionally or unintentionally disposes the link from `r` to `h`. We achieve this form of protection using *objectified link ownership*:

– Creating a dynamic link produces a `Link` object (see figure 4).
– A dynamic link can only be disposed through its corresponding `Link` object.

Note that a `Link` object represents ownership of a link – not the ownership of any particular object. Anyone with access to an object can navigate dynamic links originating from that object, but only the links' owners can dispose them.

When using traditional links, it is possible to enforce constraints on the cardinality between types. E.g. "a `Room` has exactly one `Hygrometer`". When using dynamic links, it cannot always be guaranteed that a future extension will not break such cardinality constraints. E.g. a future extension may add a second `Hygrometer`. In section 4 we will discuss a pattern that can enforce cardinality constraints in certain situations. However, as a general rule:

– Dynamic links are not constrained by type-level association. Therefore, design for "one-to-many" whenever it is practical.

We have already seen that the `context()` method is an essential part of our API. This method provides access to a simple runtime system that manages dynamic links. An overview of the runtime-system implementation is given in figure 5. The example is based on a situation where a `Room` has a single `Thermometer` and two `Hygrometers`. To improve readability, we have abbreviated the classnames used in previous examples – e.g. `Room` is abbreviated `R`.

The runtime system uses a systemwide map to associate each source object, e.g. `r`, with a corresponding *context object*, e.g. $\mathtt{c}_r$. The context object holds information about dynamic links originating from its corresponding source object. The context objects are lazily created – i.e. $\mathtt{c}_r$ is created when `context(r)` is first called. The map is a weak hash map – i.e. a context object is made eligible for garbage collection even if the global map keeps a reference to it. Consequently, developers do not have to rely on explicit link disposal – dynamic links automatically disappear when their source objects disappear.

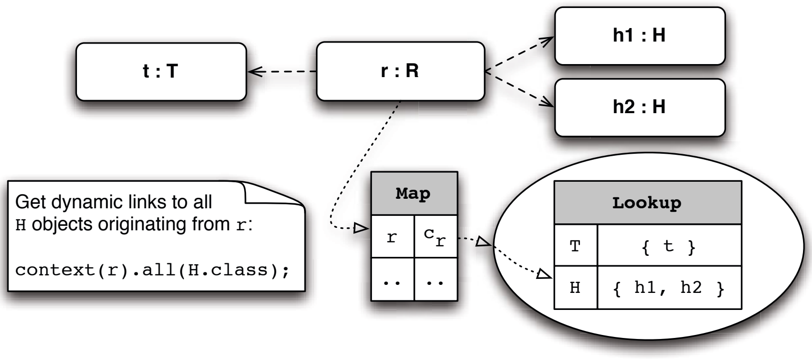


Fig. 5. Dynamic links runtime system

Each context object organizes dynamic links using a lookup. The lookup associates each destination object type, e.g. `H`, with a list of destination objects, e.g. {`h1, h2`}. Adding and disposing dynamic links correspond to changing the contents of the lookup. Obtaining links corresponds to accessing the lookup.

To summarize, let us consider evaluation of the `context(r).all(H.class)` statement by following the dotted lines in figure 5. First, `context(r)` corresponds to accessing the map and returning the corresponding context object, $\mathtt{c}_r$ – if no context object exists, it is lazily created. Second, `all(H.class)` returns a collection of all `H` objects in the context object's lookup.

Whereas obtaining and creating links is supported by methods invoked on the context object, link disposal is different. As discussed previously, link disposal happens exclusively through the `Link` object (not shown in figure 5). Thus, if the creator of a dynamic link does not keep a reference to the corresponding `Link` object, then the link – and the corresponding context information – can only disappear when the source object becomes eligible for garbage collection.

Before moving on, we would like to briefly mention a few features that space does not permit us to present in great detail:

First, it is often practical to manage ownership of groups of links that belong together – e.g. when a group of links must be disposed at the same time. Our library provides a small number of classes that support such management.

Second, fault tolerance is a crosscutting concern that may be difficult to maintain as component-based systems evolve. Our library allows for the creation of fault-tolerant dynamic links. A fault-tolerant dynamic link is a dynamic link that automatically tries to recover from a destination object's inability to satisfy its contract. This feature is motivated and inspired by [22].

4 Experience with Dynamic Links

The best evaluation of dynamic links available at the moment is experience gathered during the design, implementation, and evolution of a component-based control system for greenhouse climate control. An early version of the system was briefly mentioned in [23]. The system is currently composed of 19 components, 12 of which use dynamic links. The number of dynamic links in a running system depends on usage patterns. Normal usage easily generates more than 1,000 dynamic links, and those links may be obtained more that 500,000 times within a few minutes. The total size of the system is 12,919 lines of code.

The difficulty of anticipating required dimensions of extension, and thus enforcing the open/closed principle, is highly domain specific. In our experience, greenhouse climate control is a particularly challenging domain. First, the physical properties of individual greenhouses can be very different. Second, the set of sensors and actuators available vary greatly. Third, control requirements vary depending on the cultivar being produced and the grower's preferences.

Most research in the area of climate control has been focused on evaluating specific control strategies against various plant physiological criteria [31,30,28]. Attempts to integrate different control strategies into an extensible control system have turned out to be surprisingly difficult to perform [1].

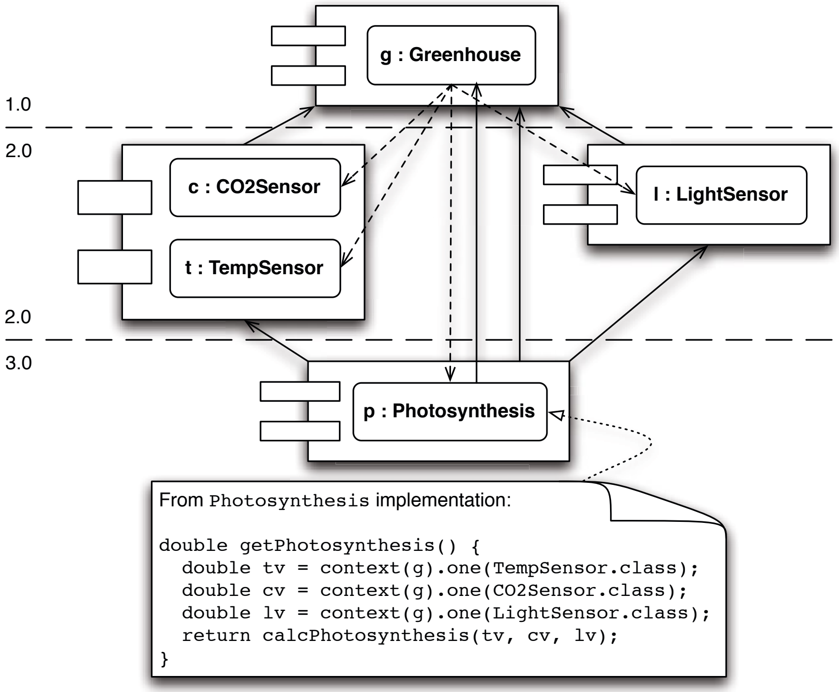
Our system is the result of a collaboration with growers, plant physiologists, and a control system vendor. We have been working on the system for two years. The concept of dynamic links has emerged during the project and plays a central role in recent versions of the system.

Figure 6 depicts selected components in the system, and some of their provided objects. For the purpose of our discussion we have organized the components in three versions – this is a simplification of the actual system's history. In the first version, a component provides `Greenhouse` objects (to improve readability only a single object is shown in the figure). In version two, two components provide CO_2, temperature, and ambient light sensors. Finally, version three adds a component that deals with photosynthesis – a measure of plant growth that can be calculated when light intensity, temperature, and the CO_2 level are known.

The figure contains two kinds of arrows: First, arrows for dependencies between components. Second, arrows for links between objects – note the difference between traditional links (normal lines) and dynamic links (dashed lines).

Based on figure 6 we will now discuss a number of concrete experiences:

– Dynamic links promote closing existing components to modification despite the presence of domain contexts, whose scopes cannot be fully anticipated.



From `Photosynthesis` implementation:

```
double getPhotosynthesis() {
  double tv = context(g).one(TempSensor.class);
  double cv = context(g).one(CO2Sensor.class);
  double lv = context(g).one(LightSensor.class);
  return calcPhotosynthesis(tv, cv, lv);
}
```

Fig. 6. Selected components in a greenhouse climate control system

In our system "a greenhouse" constitutes a domain context whose scope cannot be fully anticipated. In the broadest sense, a context is a setting in which statements may be interpreted [18]. E.g. in the context of a greenhouse we interpret statements such as "what is the temperature?" or "what is the current rate of photosynthesis?". We consider it impossible to come up with a complete list of statements that may be interpreted in the context of a greenhouse – i.e. the scope of a greenhouse context cannot be fully anticipated.

Without dynamic links, types representing domain contexts are difficult to close to modification. Addition of new context objects – e.g. `CO2Sensor` objects or `LightSensor` objects – would require modification of the `Greenhouse` type.

Note that new types of context information are not only difficult to anticipate, but can also be very different. Thus, it is difficult to extract common super types. In theory, we could use a pure tagging interface – e.g. `GreenhouseItem` – for all our unanticipated types to implement. This would actually promote closing `Greenhouse` to modification. However, since the types have very little in common, this solution would be difficult to manage for extension components, as it would often be necessary to use `instanceof` tests and typecasts to access objects using sufficiently specific interfaces.

With dynamic links, new objects can be non-invasively attached to objects of original types as the domain context they represent evolves. In figure 6, new components add links to instances of `CO2Sensor`, `TempSensor`, `LightSensor`, and `Photosynthesis`. In our system, links originating from `Greenhouse` objects refer to objects of 61 different types. Hence the `Greenhouse` type – and thus the component in which it resides – remains closed to modification.

In summary, the development style that dynamic links make possible requires developers merely to anticipate "the existence of a domain context", and "not specific dimensions of extension that must be supported by the context". Consequently, software becomes more extensible and remains closed to modification.

– Dynamic links support repeated extension, where each extension object can look up objects provided by other extensions.

It often happens that an object provided by one extension depends on objects provided by another extension. This leads to a form of repeated extension where dynamic links are used to incrementally construct a network of related objects around a common context object.

In figure 6, `p` provides the ability to calculate photosynthesis. The calculation depends on other extensions providing inputs such as temperature, `t` of type `TempSensor`, CO_2 level, `c` of type `CO2Sensor`, and light intensity, `i` of type `LightSensor`. Two things are important to note:

First, `g` represents a context in which the calculation takes place. There are many `TempSensor` objects, `CO2Sensor` objects, and `LightSensor` objects in a system. However, we need exactly those that can be found in the context of `g`.

Second, an extension can only find objects of types that are known. E.g. in order to obtain a dynamic link to `l`, it is necessary to depend on the component providing the `LightSensor` type.

The example shown in figure 6 is rather small, and thus the photosynthesis component depends on all other components being shown. A complete diagram of our system would reveal that most extensions depend only on a subset of components operating on the `Greenhouse` context – e.g. a user-interface component providing a thermometer widget needs only to know about `TempSensor` objects provided in the context of `g`. Each component may have its own incomplete view of a context, and multiple components' views may be overlapping.

In our experience, extension by attaching new objects facilitates interface segregation [14]. Dynamic links make it easy to add extension objects with "slim" interfaces, and thus clients depending on those interfaces often use all of it.

– Strive towards modeling your software so that invariants imposed by the domain do not depend on the existence of dynamic links.

The lack of class-based encapsulation makes it difficult to enforce an invariant that depends on the existence of a dynamic link. Fortunately, such invariants can almost always be avoided by taking appropriate design decisions.

In our system, we measure various values at regular intervals. The measured information is shared among components by using dynamic links originating from a `Greenhouse` object. Let us consider two different ways to implement this:

One approach is to update a measured value by replacing an object – e.g. we may dispose a dynamic link referring to an old `MeasuredCO2` object, and then create a dynamic link to a new `MeasuredCO2` object. In our experience, this implementation is often problematic, because it tends to violate invariants imposed by domain requirements. A simple invariant that may be violated is "a `Greenhouse` object must always have a `MeasuredCO2` object". Since dynamic links do not provide transaction-based creation and disposal, it is impossible to replace an object without violating the invariant. Similar problems may also emerge with more complex invariants involving more than one link.

Instead of continuously replacing a destination object, we prefer to change the state of the object. Instead of having a `MeasuredCO2` object that needs to be replaced when a new value has been measured, we use a `CO2Sensor` object that changes its internal state (see figure 6). The same `CO2Sensor` object is used throughout the lifetime of `g`. Since the state change takes place inside an object, we can enforce invariants using type-based encapsulation.

– The creation and disposal of dynamic links often coincide with creation and disposal of the object being extended. Therefore, the need for subscribing to creation and disposal events must often be anticipated.

Connecting two objects with a dynamic link – i.e. a structural extension – needs not to be anticipated by type developers. However, the need for adding new behavior to a control flow to create a link at a specific time – i.e. a behavioral extension – must be anticipated.

In our experience, the time at which a dynamic link must be created or disposed often – but not always – coincides with the time of creation and disposal of the object being extended. This is particularly the case when following our previous advice: When state changes take place inside referred objects – and not as creation/disposal/replacement of dynamic links – there is a tendency for referred objects to be created and disposed together with the object they extend.

In our system, the component responsible for managing `Greenhouse` objects makes it possible for extension components to be notified, when a `Greenhouse` object is created or disposed. We implement this using the observer pattern [8].

In our experience, the code needed to facilitate the required event notification is rarely subject to modification, even though it must once be anticipated. Thus, in practice our ability to close components to modification is rarely compromised.

Though it often happens, it is not always the case that dynamic links are created and disposed together with the object being extended. In some architectural styles an object may take the role of a message that is being passed around – e.g. pipes and filters [24]. In such cases each component handling a message may add new information using a dynamic link. In such designs a message may have a significantly longer lifetime than dynamic links used to extend it.

Our system reveals another exception from the general trend. The dynamic class-loading capability of Java allows our control system to have a software updating mechanism that can add components, while the system is running. When adding a new component, it is often necessary to add "new things" to greenhouses – i.e. new dynamic links are created, and they get to originate from `Greenhouse` objects that already exist. In our system, the component managing `Greenhouse` objects is responsible for organizing this.

– The lack of associations between types makes it important to document *sharing* and *co-existence* semantics when declaring types intended to be used with dynamic links.

The public part of a traditional type-based association manifests itself as type members – e.g. accessor and modifier methods. The names and documentation of these type members informally document the contract of that association. When there is no explicit association – as it is the case when using dynamic links – this form of documentation is not available. Consequently, the type of a destination object must provide documentation that is usually not needed or less important. In our experience, two aspects are particularly important to document:

First, a normal accessor method indicates whether ownership of returned objects is transferred to the caller – e.g. `Stack.pop()` – or if the returned objects are shared with the callee – e.g. `Stack.peak()`. With dynamic links, referred objects will almost always be shared. It is therefore important to document what happens when multiple independent units of code navigate the same dynamic link, and thus share access to a common destination object. E.g. consider the potential destination type `interface GreenhouseWindow { setOpen(boolean v); }`. `GreenhouseWindow` is probably not very useful to a ventilation component that wants to open the window for 30 minutes, since shared access enables another component to override the decision. Thus, developers should keep sharing in mind when designing and documenting types such as `GreenhouseWindow`.

Second, an association may document the roles of participating objects. With dynamic links, all objects of the same type have the same role. When multiple objects are referred to by dynamic links originating from a common source object, then we may say that they co-exist. It is important that the semantics of such co-existence is documented when declaring types of destination objects. E.g. when "a `Greenhouse` has multiple `TempSensors`", then an association may assign roles to each `Thermometer` object, e.g. "near plants" or "near the ridge". With dynamic links, co-existing `TempSensor` objects all have the same role. Thus, the `TempSensor` type must be declared, so that it makes sense to have co-existing instances. When this cannot be done, it is sometimes necessary to promote roles to types, e.g. to distinguish `PlantTempSensor` from `RidgeTempSensor`.

– *Cardinality constraints* can indirectly be achieved by limiting access to constructers of destination types. Cardinality constraints cannot be combined with abstraction.

So far, we have assumed that it is impossible to impose cardinality constraints between two types when using dynamic links. While this is to some extent true, an observation deserves to be mentioned: It is possible to indirectly impose cardinality constraints by declaring a type that cannot be instantiated directly by third-party classes or components.

Looking at figure 6, let us assume that we want to enforce that "a `Greenhouse` has exactly one `LightSensor`". We can do this by preventing subclassing – i.e. declaring `LightSensor` to be `final` – and prohibiting other components from instantiating `LightSensor` objects – i.e. making all `LightSensor` constructors private. New `LightSensor` objects can now only be instantiated by the component providing the `LightSensor` type. Thus cardinality constraints maintained by the providing component cannot be violated by other components.

Note that this technique has an important limitation: It cannot be combined with abstraction across component boundaries. In other words, the component that enforces a cardinality constraint must also be the component that provides an implementation of the destination type.

Also note this pattern's similarity with the singleton pattern [8] – both patterns prevent direct third-party instantiation.

In summary, it is our experience that dynamic links have the potential to promote the open/closed principle. To realize this potential it is important that programmers understand the benefits and limitations that dynamic links have to offer. We consider the experience presented here as a valuable starting point.

5 Related Work

The mechanisms by which dynamic links are created, obtained, and disposed are similar to the mechanisms by which objects are registered, discovered, and unregistered when using the lookup pattern [11]. The original motivation for the lookup pattern was the ability to discover distributed objects. Similarly, dynamic links can be used to discover objects provided by other components.

Some systems use lookups not merely to facilitate discovery, but to represent domain contexts with scopes that often change due to software evolution or software configuration. When used in this way, a system typically has many lookup instances, each representing something from the domain – e.g. a user, a company, or a greenhouse. The NetBeans Rich Client Platform was one of the first projects to use lookups successfully for this purpose [5]. It uses lookup instances to represent concepts such as folders (in file systems), projects, and nodes (in tree views). In such a system, lookups are only used to model selected domain contexts. With dynamic links, similar capabilities are available for any object in the system without any explicit introduction of the lookup pattern.

An approach to closing types to modification is to model an unanticipated association as a type in its own right – i.e. to use an association class [7]. Using this approach, "a `Room` has a `Thermometer`" can be modeled as "a `RoomThermometer-Association` has a `Room` and a `Thermometer`". While this approach is capable of avoiding modifications, it does involve quite a bit of unintuitive boilerplate code for declaring association classes and for managing association objects.

Another way to externalize associations is object-oriented support for relations [21] – a first-class concept inspired by the entity-relationship model used in database theory. Relations were not designed with the open/closed principle and independent components in mind. Therefore, no link-ownership mechanisms are discussed. This is, however, a prerequisite for independent extensibility. While relations as first-class concepts have attractive properties, we prefer a library-based approach, as it is easier to integrate with mainstream languages.

AspectJ [10], MultiJava [6], and many dynamic languages [9,29] support the addition of new fields and methods to existing types. We have previously noted that this form of modification compromises independent extensibility. A similar criticism can be found in [20] and [25].

Classboxes [4] also support the addition of new fields and methods, but their visibility is limited to a well-defined scope – i.e. a classbox. This makes it possible to introduce extensions to existing types without affecting existing code. Thus – like dynamic links – classboxes allow for the introduction of links to objects of unanticipated types without breaking clients of existing types. Classboxes is a language extension, while support for dynamic links is provided by a library.

6 Conclusion

Dynamic links can connect objects of unrelated types. This makes it possible to introduce links from objects of existing types to objects of unanticipated types without imposing any modifications.

Dynamic links promote extension that is compliant with the open/closed principle: First, software components become open towards new dimensions of extension – objects of new types can be freely attached to objects of existing types. Second, software components remain closed to modification – no introduction of fields and methods on existing types is required.

It is possible to implement dynamic links as a library for any mainstream object-oriented programming language. We have presented Decouplink for Java – no extension of the language or runtime is required to use it.

Dynamic links increase the design space for extensible software. We have used dynamic links to design and maintain a component-based system in a domain where dimensions of extension are difficult to predict – a climate control system for greenhouses. We have presented experience gained from this effort.

We believe that dynamic links have the potential to improve extensibility of a wide variety of software systems. We are, therefore, very much interested in experience from other domains. In particular, we would like to learn more about the long-term effects of evolving software using dynamic links. Finally, we would like to explore IDE-support that makes programming with dynamic links easier.

References

1. Aaslyng, J., Lund, J., Ehler, N., Rosenqvist, E.: IntelliGrow: A Greenhouse Component-Based Climate Control System. In: Environmental Modelling & Software, vol. 18(7), pp. 657–666. Elsevier, Amsterdam (2003)
2. Arnold, K., Gosling, J., Holmes, D.: Java Programming Language. Addison-Wesley Professional, Reading (2005)
3. Bennett, K.H., Rajlich, V.T.: Software Maintenance and Evolution: A Roadmap. In: Proceedings of the Conference of the Future of Software Engineering, pp. 73–87 (2000)
4. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: OOPSLA 2005 – ACM Sigplan Notices, vol. 40(10), pp. 177–189 (2005)
5. Boudreau, T., Tulach, J., Wielenga, G.: Rich Client Programming: Plugging into the NetBeans^{TM} Platform. Prentice Hall PTR, Englewood Cliffs (2007)
6. Clifton, C., Leavens, G., Chambers, C., Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In: OOPSLA 2000 – Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 130–145 (2000)
7. Fowler, M.: UML Distilled. Addison-Wesley Professional, Reading (2004)
8. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)
9. Goldberg, A., Robson, D.: Smalltalk-80: The Language and its Implementation. Addison-Wesley Longman Publishing, Amsterdam (1983)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
11. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management. Wiley, Chichester (2004)
12. Lehman, M.: Programs, Life Cycles, and Laws of Software Evolution. Proceedings of the IEEE 68, 1060–1076 (1980)
13. Lehman, M., Ramil, J.: Software Uncertainty. In: Software 2002: Computing in an Imperfect World, pp. 477–514 (2002)
14. Martin, R.C.: The Interface Segregation Principle. C++ Report (1996)
15. Martin, R.C.: The Open-Closed Principle. C++ Report (1996)
16. Meijer, E., Szyperski, C.: Overcoming Independent Extensibility Challenges. Communications of the ACM 45(10), 41–44 (2002)
17. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1988)

18. McGregor, J.: Context. Journal of Object Technology 4(7), 35–44 (2005)
19. Object Management Group: OMG Unified Modeling Language^{TM} (OMG UML), Infrastructure, `http://www.omg.org/spec/UML/2.2/`
20. Ostermann, K., Kniesel, G.: Independent Extensibility – An Open Challenge for AspectJ and Hyper/J. In: ECOOP 2000 – Workshop on Aspects and Dimension of Concerns (2000)
21. Rumbaugh, J.: Relations as Semantic Constructs in an Object-Oriented Language. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, pp. 466–481 (1987)
22. Rytter, M., Jørgensen, B.N.: Enhancing NetBeans with Transparent Fault Tolerance. Journal of Object Technology 9(5) (2010)
23. Rytter, M., Jørgensen, B.N.: Composing Objects in Open Contexts using Dynamic Links. In: Informatics – Software Engineering and Applications (2010)
24. Shaw, M., Garlan, D.: Software Architecture – Perspectives on an Emerging Dicipline. Prentice-Hall, Englewood Cliffs (1996)
25. Steimann, F.: The Paradoxical Success of Aspect-Oriented Programming. In: OOPSLA 2006 – Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, pp. 481–497 (2006)
26. Szyperski, C.: Independently Extensible Systems – Software Engineering Potential and Challenges. In: Proceedings of the 19th Australasian Computer Science Conference (1996)
27. Szyperski, C.: Component Software – Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Professional, Reading (2002)
28. Tantau, H., Lange, D.: Greenhouse Climate Control: An Approach for Integrated Pest Management. Computers and Electronics in Agriculture 40, 141–152 (2003)
29. Thomas, D., Hunt, A.: Programming Ruby: A Pragmatic Programmer's Guide. Addison-Wesley Professional, Reading (2000)
30. Van Pee, M., Berckmans, D.: Quality of Modelling Plant Responses for Environment Control Purposes. Computers and Electronics in Agriculture 22, 209–210 (1999)
31. van Straten, G., Challa, H., Buwalda, F.: Towards User Accepted Optimal Control of Greenhouse Climate. Computers and Electronics in Agriculture 26, 221–238 (2000)