Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering, © JOT 2010
Online at http://www.jot.fm.

Enhancing NetBeans with Transparent Fault Tolerance Using Meta-Level Architecture

Martin Rytter[a] Bo Nørregaard Jørgensen[a]

a. The Maersk Mc-Kinney Moller Institute, University of Southern Denmark

Abstract

In component-based systems, fault-tolerance concerns are typically handled by manually programmed fault containers. The purpose of fault containers is to prevent error propagation across component boundaries by means of redundant service providers. However, manually programmed fault containers are often subject to evolutionary pressure when components change. In this paper we present a meta-level architecture that eliminates the need for manually programmed fault containers. The meta-level achieves fault tolerance using dynamic fault containers, thereby reducing evolutionary pressure. We present an implementation and evaluation of our approach in context of the NetBeans Rich Client Platform.

Keywords fault tolerance, software evolution

1 Introduction

The main benefit of adding fault tolerance to component-based software systems is that they become more reliable, because they can continue to operate in the presence of faults or even partial failure. Unfortunately, maintaining the fault-tolerance logic of such systems often results in evolutionary pressure on components that would not need to change if the fault-tolerance concern was not present. While we cannot eliminate evolutionary pressure on entire systems, it is desirable to reduce the number of software components that will be affected [Leh80]. This is particularly important when considering software components to be units of independent production, acquisition and deployment [Szy02].

An illustrative example is the fault-container pattern [Sar03]. The fault-container pattern is a variant of the adapter pattern [GHJV95] providing a fault-tolerant interface to unreliable objects. The adapter in this variant of the pattern is called a fault container. The fault container achieves fault tolerance by performing error detection and recovery. A simple recovery strategy is dynamic redundancy: The idea is to

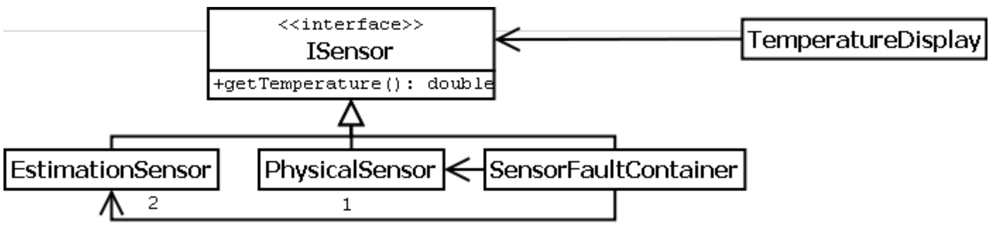

Figure 1 – A fault container for a sensor.

sequentially select among multiple redundant providers of the service-provider interface. As long as one of them works, a fault can be contained.

However, the fault-container pattern tends to be prone to evolutionary pressure when clients and service providers change. To see why, let us consider the simple example in figure 1. In the example we have a service-provider interface, `ISensor`, with a single method, `getTemperature()`, returning the temperature in a building. In our pursuit of fault tolerance we also have a `SensorFaultContainer` implementing the `ISensor`-interface. Let us assume that we have a client, `TemperatureDisplay`, that simply displays the temperature to a user.

To see how evolutionary pressure on `SensorFaultContainer` emerges, let us consider adding new providers of `ISensor`. First, let us provide a `PhysicalSensor` that can return a temperature by measuring it. To make this new functionality available to clients that require fault tolerance, we need to modify `SensorFaultContainer`. This is because `SensorFaultContainer` depends on all providers of `ISensor` in order to implement the fault-tolerance concern. Therefore, addition of a new provider imposes evolutionary pressure. Second, let us provide an `EstimationSensor` that can estimate the temperature by inspecting the state of a heating valve. We prefer using `Physical-Sensor`, but in case of failure `EstimationSensor` will do. This decision requires yet another modification of `SensorFaultContainer`, such that the preferred sensor is selected if it is available. We observe from this example, that fault containers easily become prone to changes during software evolution.

In monolithic software systems evolutionary pressure on fault containers may be manageable, as the codebase is under control of a single organization. However, in component-based systems built from independently developed components, the situation is different. In such systems, manually programmed fault containers must obviously reside in components. These components will be subject to evolutionary pressure when removing, adding or modifying components containing providers of the fault-tolerant service-provider interface. The inevitable evolutionary pressure on fault containers makes the ideal of components as units of independent production, acquisition and deployment difficult to achieve.

In this paper we propose a novel approach for achieving fault tolerance in component-based systems. The idea is based on moving fault-tolerance concerns away from base-level components and into a meta-level. Thus, programmers do not have to write fault containers manually. Instead, dynamic fault containers are provided transparently by the meta-level. This transparency is achieved by letting clients obtain required references using the lookup-service pattern [KJ01, KJ04]. Certain fault-tolerance decisions are domain specific and can never be fully automated. To address such decisions our meta-level is open to configuration by the base-level. Using our approach, it is possible to minimize evolutionary pressure due to fault-tolerance concerns.

We have implemented a prototype of our idea on top of the NetBeans Rich Client Platform [BTW07]. We have chosen this platform because its use of the lookup-service pattern allows for easy integration with our approach. It also allows us to validate the compatibility of our approach with an industry-strength component platform.

Our work indicates that reduced evolutionary pressure can be achieved when using our approach, due to its strict separation of fault-tolerance concerns from application logic. We expect this insight can improve the maintainability of component-based software systems with requirements for fault tolerance.

The paper is structured as follows. Section 2 presents state of the art. Section 3 revisits the introductory example in order to motivate properties of fault-tolerant, evolvable, component-based systems. Section 4 presents an overview of our approach. Section 5 presents our prototype. Section 6 provides an evaluation. Section 7 concludes and summarizes our findings.

2 State of the art

In this paper we use the taxonomy of dependable systems given in [ALRL04]. In this taxonomy threats to dependability of software systems include faults, errors and failures. A failure is a transition from correct to incorrect service. An error is the part of the total system state that may lead to failure. A fault is the adjudged or hypothesized cause of an error – i.e. a development fault (a bug) or a physical fault (a hardware problem). Fault tolerance is the idea of achieving dependability through avoidance of service failures in the presence of faults. Fault tolerance is thus different from other means of achieving dependability, e.g. fault prevention that tries to prevent the introduction of faults.

The idea that a reliable system can be built from unreliable components has been around for a long while [vN56]. In the early days of computing, fault tolerance dealt primarily with unreliable hardware components. Success in this area inspired researchers to apply similar ideas to software – i.e. dynamic redundancy in the form of recovery blocks [Ran75] and masking redundancy in the form of N-version programming [CA78]. In retrospect most progress came at the price of increased complexity [RX94]. The use of N-version programming and recovery blocks was therefore limited to applications with high fault-tolerance requirements [AB87].

Some of the added complexity comes from the broader problem of dynamically reconfiguring a running application while preserving consistency [KMSL83, KM90]. One of the keys to success in this area is modularity. If a system is not modular it is hard to isolate a component and replace it at runtime. The problem becomes even harder when state exists that must be preserved. Interestingly, this important insight seems to have emerged independently in the areas of fault tolerance and dynamic reconfiguration [Gra85, KM85]. Modularity helps us change systems at runtime, in much the same way it helps us change them during software evolution.

The advent of object orientation and reflective meta-level architectures led to rethinking the existing fault-tolerance techniques in a new context [XRRCS94, XRZ]. These ideas could potentially help solve the important problem of supporting development of fault-tolerant systems without greatly complicating implementation, readability, and maintenance [RX94]. In the view of the authors, this has only happened on a relatively small scale – mostly in the form of design patterns such as the fault-container pattern [Sar03], or as libraries concerned with specific forms of failure [FN]. No truly general and reusable fault-tolerance mechanism has gained wide acceptance.

Instead, the concept of exceptions has evolved as a general form of "fault-tolerance light"-mechanism that is now present in most languages [GRRX01]. Whereas exceptions provide means for error signaling, and separation of normal and exceptional behavior, they normally do not provide means of recovery. Thus, exceptions are limited to provide useful infrastructure that supports certain aspects of fault tolerance, i.e. controlled error propagation.

There has been attempts to expand the role of exceptions using the resumption model. In this model continuation of control flow can be controlled from inside exception handlers. This model could in theory enhance the fault-tolerance capabilities of exceptions. However, while the model is flexible, it is also difficult for programmers to use [GRRX01]. Most programming languages, therefore, adopt the traditional termination model [AGH05]. The Ruby language does support the resumption model, but using it is generally discouraged [TH00]. The main benefit of the termination model is that the model itself is simple. When resumption (or recovery) is a domain requirement, this leads to complicated code no matter which of the two models are used. Another means of achieving a more flexible exception model is through a meta-level architecture [GBR99].

One particular area of fault tolerance is distributed systems. Here separation of application concerns and fault-tolerance concerns tends to be handled using message-oriented middleware [HW03]. This trend relies on various forms of client transparency which can be provided using the lookup pattern [KJ01, KJ04]. One example is location transparency where a client is not aware of the location of a server. Another example is fault-tolerance transparency, where a client is not aware of fault-tolerance mechanisms present in the middleware [FN]. These forms of transparency limit evolutionary pressure on clients.

We believe that the main challenge for fault tolerance in component-based systems is to achieve low evolutionary pressure among components, without sacrificing the ability to tolerate faults. The use of client transparency in distributed systems achieves low evolutionary pressure on clients. However, in component-based systems we must achieve low evolutionary pressure for all components, both client components that require fault-tolerant access to services, and provider components that implement unreliable services. It is therefore required that the use of fault-tolerance transparency is expanded to include provider components. Solutions that reduce this gap are essential to promote the use of fault tolerance in component-based software development.

3 An example

In this section we emphasize properties facilitating fault tolerance in component-based software systems subject to software evolution. The properties are motivated by expanding our example introduced in section 1.

Let us consider the `ISensor`-interface from figure 1 being part of the simple component-based control system shown in figure 2. The purpose of the system is to control the temperature in a building using an actuator and a sensor. Requirements of our example system are enumerated below:

- The system must be fault tolerant with respect to sensor failures.
- A user-specified temperature set point must be maintained when possible.

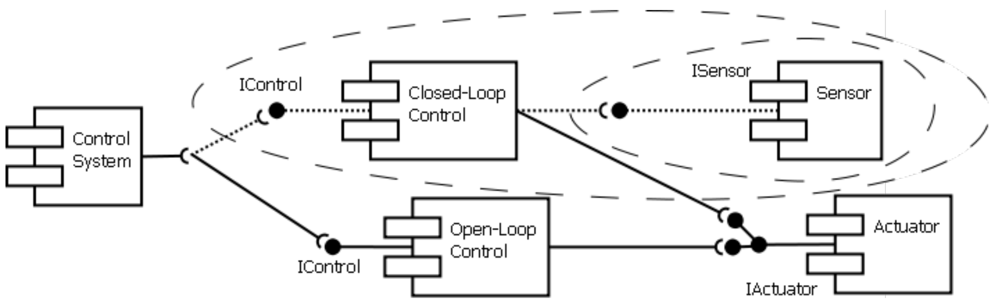

Figure 2 – Control system with fault-tolerance requirements.

- To prevent frost, temperatures below zero must be avoided.

The system is composed of five components. The sensor component provides measurement of the current temperature. The actuator component provides functionality to increase the temperature using a heating system. The open-loop and closed-loop components provide two different control strategies. With closed-loop control, the difference between the user-specified set point and the measured temperature is used to control the actuator. With open-loop control, only the user-specified set point is used. Closed-loop control is preferred over open-loop control because feedback from the sensor allows for more precise temperature control. Open-loop control is not very precise, but it works without a functioning sensor, and it is good enough for frost protection. The control-system component drives the system.

A provider component, e.g. `Sensor`, provides a set of provider objects. A client component, e.g. `ClosedLoopControl`, requires a set of provider objects. A provider object implements a service-provider interface, e.g. `ISensor`. A particular component, e.g. `ClosedLoopControl`, may assume the roles of client component and provider component simultaneously. We have indicated potential failures using dotted lines for connectors in our diagram. The sensor component may fail to provide its service. The diagram illustrates that such failure may lead to error propagation along connectors.

Our example is designed to support fault tolerance using the fault-container pattern. Fault tolerance is possible when an erroneous provider object can be safely substituted by another at runtime. We think of such substitution as a simple form of dynamic reconfiguration. The innermost dashed circle indicates an area of the system in which a sensor failure cannot be contained. Errors can be contained within the outermost circle because closed-loop control is not the only provider of `IControl`. By dynamically reconfiguring the system to use open-loop control, we achieve degraded behavior, i.e. frost protection. When the sensor once again starts to work, another dynamic reconfiguration can restore closed-loop control behavior.

For the sake of simplicity, we assume service-provider interfaces such as `ISensor`, `IControl` and `IActuator` to be declared in components not shown in the diagram. To illustrate problems with evolutionary pressure, we also assume a traditional implementation in which fault containers reside inside client components. Therefore, in the example, the control-system component is responsible for detecting control algorithm failure, and containing the fault by choosing an alternative algorithm. Similarly, the closed-loop component is responsible for detecting sensor failure and propagating the fault.

We wish to highlight that a design based on the traditional interpretation of fault-container pattern is prone to evolutionary pressure towards client components.

Too see why, consider the following change requests:

- *Add another sensor component to provide redundancy:* With two providers of `ISensor`, failures can now be contained when the other sensor is still working. If the possibility of adding an additional sensor was not foreseen in the implementation of the client's fault container, this change is likely to require modification of the fault container for `ISensor` to support selecting an alternative sensor, in case the active sensor fails. Thus, evolutionary pressure on the closed-loop component is expected. Note that this change request is similar to the example given in section 1.

- *Add logger component that saves a history of sensor values:* The new logger component needs fault-tolerant access to sensors. This is already available as a fault-container instance controlled by the closed-loop component. Sharing this instance with other client components is likely to require modifications in the closed-loop component. Alternatively, the new logger component might duplicate the fault-tolerance functionality by copying the fault container of the closed-loop component.

Conceptually, both change requests are additive. We simply add a component without any desire to modify existing functionality. However, evolutionary pressure emerges anyway due to fault-tolerance concerns. In case of the first change request, the fault-tolerance logic itself needs to change to allow dynamic selection of a new provider. In case of the second change request, the existing fault-container instance needs to be shared if redundant fault-tolerance functionality is to be avoided. Redundant fault-tolerance functionality is undesirable as it accelerates the maintainability problems caused by evolutionary pressure on fault containers. Both change requests tend to require modification of the component providing the fault container – for both change requests this is the closed-loop component. The two forms of evolutionary pressure can be avoided when two properties are satisfied:

- *Transparent fault tolerance:* The need to modify existing fault-tolerance logic leads to no evolutionary pressure on neither client components nor provider components.

- *Transparent sharing:* The need to obtain, introduce, or remove references to provider objects participating in fault tolerance leads to no evolutionary pressure on neither client components nor provider components. Transparent sharing is a broader concept than location transparency, because it also provides transparency between co-existing providers.

Transparent fault tolerance is usually easy to achieve with respect to client components. For provider components the situation is different, because strong consistency guarantees during dynamic reconfiguration require participation, i.e. using state-transfer functions [WS96]. It is, therefore, normal to relax fault-tolerance transparency to the client side.

Transparent sharing is easier to achieve – at least in theory. Using a lookup service, client and provider components can register and obtain references to service-provider objects independently of each other. All that is needed is a common service-provider interface. In practice, the primary obstacle to transparent sharing is design and not mechanism. The designers must predict useful sharing contexts [McG05] (i.e. lookups) and contracts for objects that makes sense to share (i.e. interfaces) [BTW07]. However, this is a general design problem that is not particular to fault tolerance.

In summary, with transparent fault tolerance and transparent sharing, evolutionary pressure among components is reduced. This applies not only to client components, but to all components containing clients or service providers.

4 Overview of the approach

Our approach provides support for fault tolerance that is compatible with state-of-the-art component platforms. We argue that the common interpretation of the fault-container pattern is sufficient for fault tolerance but that it, in its current form, tends to hinder evolution of individual components. The limitation of the common interpretation is that it causes evolutionary pressure on components containing manually programmed fault containers.

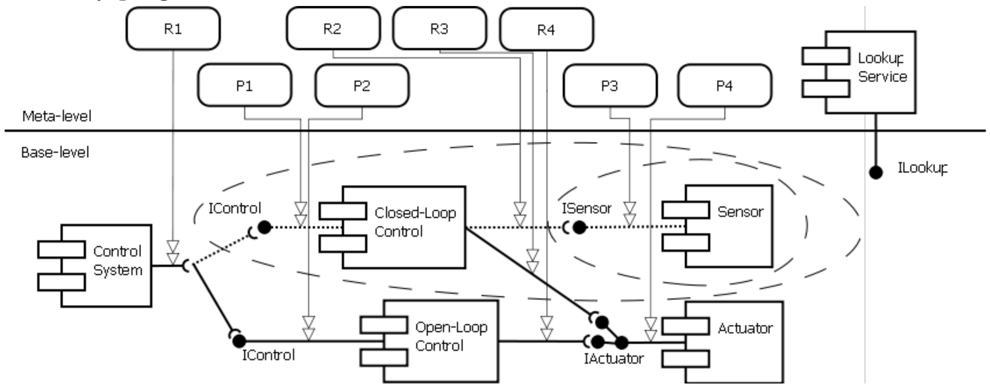

Figure 3 – Control system with meta-level fault containment.

Our approach proposes a dynamic fault container. A dynamic fault container is not statically defined by a programmer, it is created dynamically during runtime composition. Thus, dynamic fault containers are not instantiated explicitly by clients, instead client components in the base-level obtain references to fault containers when they obtain a provider reference through a lookup service [KJ01, KJ04].

4.1 Structure

The combination of a lookup service and a dynamic fault container can be perceived as providing a meta-level concerned with fault tolerance. Figure 3 shows our example system augmented with a fault-tolerance meta-level.

The diagram illustrates that fault containers are no longer provided by components. Instead, dynamic fault containers are meta-objects provided by the meta-level. A dynamic fault-container instance exists for each required reference in the base-level, i.e. $\{r_1, r_2, r_3, r_4\}$. Similarly, a meta-object exists for each service provider object, i.e. $\{p_1, p_2, p_3, p_4\}$. The dynamic fault containers implement a meta-object protocol, providing required interfaces to base-level objects, by forwarding to providers.

All the meta-objects are associated with a lookup instance (dependency lines are not shown on the diagram to improve readability). The lookup service provides an explicit interface to the base-level. The interface allows for obtaining required references. It also allows the base-level to associate candidate service providers with a lookup instance.

Dynamic fault containers implement both components of fault tolerance, namely error detection and fault repair. Error detection is implemented by validating runtime behavior against the contract specified in the service-provider interface. Fault repair is provided using dynamic resolution of required references. For example, in figure 3, r_1 can be resolved by forwarding to p_1 or p_2. r_2 can only be resolved by forwarding to p_3. p_4 is shared because both r_3 and r_4 can forward to it.

Whereas the interface to the lookup service is explicit, this is not the case for dynamic fault containers. Dynamic fault containers provide an implicit interface to the meta-level [Zim96]. The base-level obtains a reference to an object of a type present in the base-level through the lookup service, but it remains oblivious of the fact, that the returned object implements a meta-object protocol for fault tolerance. The client components in the control system are unaware of the existence of dynamic fault containers. I.e. the `ControlSystem` component only knows that it has obtained a reference to an `IControl`-object.

In general, successful fault containment cannot be guaranteed, as it depends on the availability of appropriate providers. The meta-level must, therefore, consider error propagation. In our example, a sensor failure cannot be contained by r_2, because no alternative sensor is available. Errors are therefore propagated and eventually contained by r_1 by a dynamic reconfiguration from p_1 to p_2.

4.2 Error detection

Traditional fault containers are programmed manually. This means that error detection can be customized by the programmer while having specific clients and providers in mind. This is not possible with dynamic fault containers. Dynamic fault containers must implement error detection exclusively based on an explicitly declared service-provider interface. As we shall see, this restriction has important implications.

From the perspective of our closed-loop control algorithm in figure 3, it is reasonable to consider a malfunctioning sensor as an error – we do not expect this event to happen, and if it does, we can do nothing about it. Consider adding a surveillance component that automatically sends an email when a sensor is not working. To this new component, a malfunctioning sensor is not an error but an important state, which the interface of a sensor has to provide access to. In general, what is considered an error in one context may be an expected state in another.

In our approach the notion of normal behavior must be declared in the service-provider interface. We expect exceptions declared in a service-provider interface to be normal behavior, while undeclared exceptions are considered failures. Consequently, client-specific needs must be expressed by requiring different interfaces. This principle encourages programmers to put customization of error semantics in adapters [GHJV95] instead of placing those customizations in client code.

Note that a declared exception is not always a checked exception – an unchecked exception may, in fact, be declared, though it is not the common case. A checked exception is an exception type that the compiler forces clients to address (handle or rethrow). Declaring an exception indicates that occurrence of the exception is normal behavior in a given context (i.e. during a method call).

Using declared and undeclared exceptions to distinguish normal and erroneous behavior can be made subject to discussion. However, following the general recommendations for using exceptions our decision is reasonable. [ZHR$^+$06] encourages use of declared exceptions for problems that clients are expected to resolve, while undeclared exceptions should be reserved for irresolvable problems. Introducing our approach, this translates to using declared exceptions for problems that should always be propagated to clients, while undeclared exceptions may be contained by a fault container at the meta-level.

Our prototype has been implemented in Java [AGH05]. In Java there is a clear separation between declared and undeclared exceptions. In languages where this is not the case, another means of distinguishing normal behavior from erroneous behavior must be adopted. We encourage this decision to be taken with the exception model of the language in mind [GRRX01].

There exist contracts between components above the syntactic level, which are usually not enforced by compilers or even at runtime [BJPW99]. Examples include pre- and post-conditions, synchronization contracts and quality of service contracts. Tools for these purposes can interoperate with our meta-level architecture by implicitly (from the perspective of the programmer) throwing an undeclared exception, when a contract is violated.

4.3 Fault repair

When an error has been detected, the meta-level attempts fault repair. If fault repair is successful the meta-level achieves fault containment, and thus isolates the problem from clients in the base-level.

Fault repair is implemented using an algorithm for dynamic selection of service providers. The algorithm decides how required interfaces are resolved by dynamic fault containers. This translates to performing a successful method-call forwarding to the best provider from those available under the given conditions.

We have argued that error detection can be based exclusively on the provided interface, i.e. the contract. On the contrary, fault repair can only be treated in this general way, when the observable behavior of all available implementations are equally desirable to their clients. In other words, when their quality of service is the same.

In our control-system example, r_1 requiring `IControl` can be satisfied using p_1 providing closed-loop control or p_2 providing open-loop control. These are not redundant. Closed-loop control is superior and should be the first choice. Open-loop control provides degraded, but reliable, behavior. Note that this decision is domain dependent and can never be completely avoided. However, we can avoid implementing the decision explicitly in clients or providers, thus avoiding evolutionary pressure in case the preferred choice needs to change.

In our approach, the selection semantics for deciding which provider to choose is expressed by parameterizing lookup-service instances with a content-provider object. A lookup service uses its content-provider object to determine the order in which it should try service-provider objects implementing the requested interface. In this approach, the system composer determines the selection strategy – clients or providers are not involved. This approach is comparable to many of the ideas in the area of externalized adaption [GS02].

A content-provider object simply provides the order in which to try a set of provider objects. Examples include a content-provider object that tries provider objects in a list, a tree or any other ordered data structure. The set of provider objects may be hardcoded by a programmer, or it might reflect a user decision. All dynamic fault containers returned from a given lookup instance uses the content-provider object of that instance. This yields a simple, yet general, fault-repair strategy:

1. Get a prioritized list of service-provider objects from the content-provider object.
2. For each object in the list, try to forward the call. In the case of an error (undeclared exception), contain the fault and continue iteration. If successful, break the loop.
3. If no functional object was found, propagate an error to the next fault container on the execution stack using an undeclared exception. If a functional object was found, pass on the returned value (or void).

The algorithm simply provides fault repair by dynamically trying all suitable providers. In our previous example, r_1 might first attempt method-call forwarding to p_1. In the case of an error, forwarding to p_2 allows for successful repair and thus fault containment using degraded behavior.

Fault repair, and thus fault tolerance, cannot always be guaranteed. Even if we assume that the implementation of the dynamic fault container is correct, such guarantee cannot be given. The reason is that provider selection relies on a dynamic context of available providers, i.e. the content of a lookup service. This context is dynamic, because providers can be added to the system and removed from the system at runtime. Service-provider objects may be added or removed as components enter or leave the system, or as a consequence of normal application logic. It may also be the case that all service-provider objects in the context fail. When no suitable service-provider object is available, error propagation is unavoidable.

4.4 Error propagation

An error that cannot be recovered from must be propagated. Traditional programming languages implement error propagation using an exception model. Often such models include a number of useful features that we want our approach to be compatible with.

In our approach error propagation is a collaborative effort between the base-level and the meta-level. During propagation the meta-level attempts fault containment at each defined boundary (dynamic fault container). Between boundaries each base-level component may implement measures to preserve its own consistency, i.e. by releasing resources using `try-finally` constructs. Such activity does not require knowledge about the type of the exception being propagated, because each component is solely concerned with its own consistency.

Note that our approach to error propagation integrates nicely with existing exception models. Thus, a meta-level for fault tolerance is a supplement to traditional exception handling – not a replacement.

In our control-system example, a sensor failure may be signaled by the base-level from inside the sensor component. From here the resulting exception will be propagated out of the component until it reaches the first dynamic fault container, r_2. At this point the meta-level will attempt fault containment, which, in the depicted configuration, will fail, because no alternative sensor is available. The exception is, therefore, propagated back to the base-level and through the closed-loop component. On its way the component may perform various consistency-preserving activities using the exception model of the language. The exception will eventually reach r_1, where fault containment will be achieved by degrading to open-loop control. This graceful degradation of the system is transparent to the control system component.

4.5 Consistency

To achieve consistency during dynamic reconfiguration, our fault-tolerance approach is based on three assumptions:

- Method invocations are assumed to be atomic.
- Service-provider interfaces must assume that instances are shared among multiple clients.
- No internal object state is transferred when using dynamic reconfiguration to achieve fault repair.

These assumptions must be kept in mind when designing service-provider interfaces. In other words, it is not possible to take any existing interface and simply assume that it can be used in a fault-tolerance context. Note that restrictions on the way interfaces can be expressed already exist when designing for thread safety [Goe06]. Like thread safety, fault tolerance is a non-functional concern that may affect the design of interfaces.

As an example, consider an interface for ordering items from a web shop. Such an interface may be called `IShop` and have the methods `addItem(Item)` and `commit()`. Using this interface with our approach does not guarantee that all items will actually be in the order – the reason is that method calls may end up at different objects providing the interface. A better interface is one that takes an entire order in a single method invocation, i.e. `IBetterShop { void commit(Order); }`. Note that there is another reason why `IShop` is a bad interface. Multiple clients getting access to a shared object of `IShop` will not know exactly what they order when calling `commit()`, as the interface is not designed with multiple clients in mind. This is clearly undesirable when the interface is intended to be shared using a lookup service.

The alert reader will notice that what we did in the `IShop` example was to turn a stateful interface into a stateless interface. This is not always practical. So how do we support fault tolerance when we have state that cannot be avoided? The traditional approach is to use a state-transfer function [WS96]. As we explained in section 3, we want to avoid this solution to minimize evolutionary pressure. As an alternative, we suggest to externalize and share state, instead of transferring it during dynamic reconfiguration. We believe that externalizing and sharing state in a consistent manner is not harder or more error-prone than writing a reliable state-transfer function.

4.6 Context

In our simple control-system example, it is sufficient to use a single lookup-service instance. However, consider redesigning the system to control the temperature in multiple separate buildings. In this case we need a lookup-service instance for each building under control. Otherwise we would not know to which building an instance of `ISensor` belongs.

Modeling a domain using many dedicated lookup-service instances is a commonly used technique. Note that this technique uses lookup services to achieve more than merely location transparency. It uses a lookup-service instance to create an open context [McG05] – e.g. a building. A component introducing the "building" concept does not need to know the types of objects other components may desire to put in this open context – e.g. sensors, actuators and control strategies. [BTW07] contains several examples of when and how to use this pattern.

With our fault-tolerant lookup service we can create contexts that are not merely open to new types of objects – it also provides fault-tolerant access to those objects. As previously discussed, fault tolerance is made transparent by using dynamic fault containers as proxies for service-provider objects. Thus, a fault-tolerant lookup service has semantics different from a traditional one. To use this new style of lookup service, developers must learn to identify the need for fault-tolerant contexts. We believe this to be an important design skill during the design of any fault-tolerant system regardless of which techniques are employed.

5 Implementation

This section describes our prototype implementation based on the NetBeans Rich Client Platform. NetBeans provides a highly extensible component model (module system in NetBeans lingo). Therefore, our prototype does not require modification of the platform itself. Our prototype is simply a NetBeans component providing a new lookup service.

We have chosen the NetBeans Rich Client Platform for three reasons. First, the platform uses a service-oriented approach to component integration that already relies on the lookup pattern. Second, the platform has been used successfully in numerous long-lived projects and is therefore considered to be mature. Finally, we use the platform in a number of industrial projects, thus making it a natural choice for practical reasons.

5.1 The existing lookup service

Variants of the lookup pattern are used extensively in the NetBeans Rich Client Platform [BTW07]. Lookup services are the primary mechanism for components to connect to other components. The existing lookup-service implementation supports transparent sharing, but not transparent fault tolerance.

Transparent sharing allows a client component to obtain a reference to a service-provider object without depending on the component providing it. This is possible when a client and a provider component depend on a common service-provider interface. Such interface may be declared in a component independent of the provider component and the client component.

The context, in which sharing takes place, is a lookup-service instance. On the NetBeans Rich Client Platform it is common to use many lookup-service instances: A global lookup service provides a general context accessible to everyone. Other lookup-service instances provide access to more specific contexts such as “files in a folder”, “items in a menu” or “shortcuts to execute an action”. Specific lookup-service instances must be obtained directly or indirectly through the global lookup service.

Transparent fault tolerance is not supported by the standard lookup service. This means that a client component having obtained a reference to an erroneous service-provider object is itself responsible for error detection and fault repair. As we have already seen, this can be a source of software evolution problems.

5.2 Integrating fault containment

The API to our fault containing meta-level is similar to the existing NetBeans lookup service API. The most important classes are shown in figure 4.

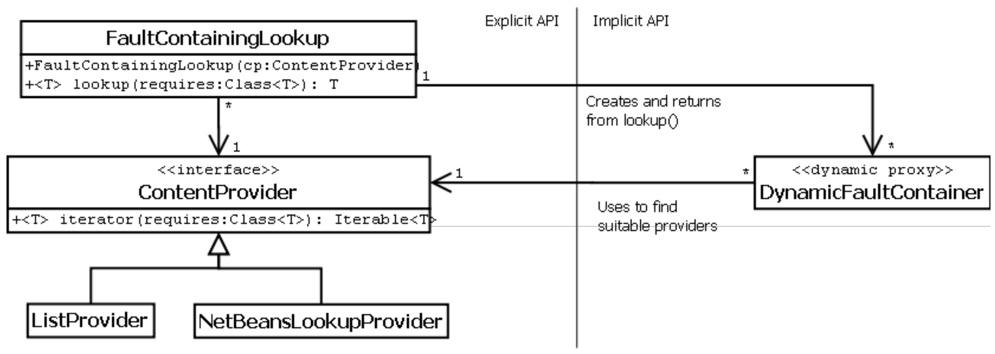
Explicit API
Implicit API
FaultContainingLookup
+FaultContainingLookup(cp:ContentProvider)
+<T> lookup(requires:Class<T>): T
1
*
1
Creates and returns from lookup()
<<interface>>
ContentProvider
+<T> iterator(requires:Class<T>): Iterable<T>
1
*
<<dynamic proxy>>
DynamicFaultContainer
*
Uses to find suitable providers
ListProvider
NetBeansLookupProvider

Figure 4 – Fault tolerance API.

An explicit API to the meta-level allows for instantiation of `FaultContaining-Lookup` objects. Each lookup service must be parameterized with a `ContentProvider`. As previously discussed (section 4.3) this enables arbitrary selection strategies to be defined. Our diagram shows two example strategies. A `ListProvider` is a simple prioritized list of providers. A `NetBeansLookupProvider` is a wrapper for a classic NetBeans lookup service.

The lookup method of `FaultContainingLookup` appears to return a service-provider object of the requested type (specified by a `Class`-object). In reality, it returns a `DynamicFaultContainer` selecting among multiple suitable service-provider objects. Thus, the `DynamicFaultContainer` becomes an implicit API to the meta-level.

The dynamic fault container is implemented as a dynamic Java proxy [jav]. The dynamic proxy performs method-call forwarding to suitable providers, detects and propagates errors. The core of the implementation is the `invoke()`-method handling all method invocations on the fault container. This method is shown in Listing 1.

Method-call forwarding inside the proxy is implemented using reflection. This approach was chosen for simplicity. We expect there is a substantial potential for performance improvements by generating fault-container classes at load time using bytecode transformation, thus avoiding the need for reflection.

A limitation of dynamic proxies in Java means that they can only implement interfaces. It is not possible to create a dynamic proxy for an object of a normal class. For the purposes of a prototype this is acceptable. However, it would clearly be desirable to support dynamic fault containment for any type – not just interfaces. This could similarly be achieved by automatically generating fault containers for all types using bytecode transformation at class load time.

Java supports separation between declared and undeclared exceptions. Our error detection mechanism is therefore identical to the approach described in section 4.2. That is, declared exceptions are considered normal behavior and undeclared exceptions are considered erroneous.

Error propagation on the base-level is implemented using unchecked exceptions. When we have to signal an error because no provider was available or because all available providers failed, we throw a special purpose `ServiceUnavailableException`.

A consequence of providing fault tolerance as a NetBeans component is that the global lookup-service instance is not adapted. We do not expect this to be a problem in practice though. Programmers can always establish a new lookup-service instance and

```
public class DynamicFaultContainer implements InvocationHandler {
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    // Try invoking candidate services until one succeeds.
    Iterable<?> candidates = content.iterator(required);
    for(Object s : candidates) {
      try {
        return method.invoke(s, args);
      } catch(InvocationTargetException x) {
        // Check to see if exception matches declaration.
        Throwable cause = x.getCause();
        for(Class declaredException : method.getExceptionTypes()) {
          if(cause.getClass().isAssignableFrom(declaredException)) {
            // It does, rethrow cause.
            throw cause;
          }
        }
      }
    }
    // No working service was found. Error propagation unavoidable.
    throw new ServiceUnavailableException(required, method);
  }
}
```

Listing 1 – Dynamic fault-container implementation.

provide it – directly or indirectly – through the global lookup service. As previously mentioned, it is already a well-established technique to create vast amounts of special-purpose lookup services when using the NetBeans Rich Client Platform. Lookup services are used for modeling open contexts and not just for obtaining references to objects in other components. Thus, our fault-tolerant lookup service is a natural extension of this idea.

6 Evaluation

We will now present a brief evaluation of our approach to meta-level fault tolerance. The evaluation is based on a comparison between two situations:

The first situation is manual fault tolerance. In this situation fault containers [Sar03] are manually programmed when they are needed, or when their need is anticipated. When a fault container has been programmed, it is distributed in a component chosen freely by the developer. We believe this situation to resemble common practice when designing fault-tolerant, component-based systems.

The second situation is our approach to meta-level fault tolerance. In this situation programmers do not program fault containers explicitly. Instead, they must anticipate where to use fault-tolerant lookups and possibly provide a domain-specific content-provider object. Fault tolerance is transparently handled by dynamic fault containers.

In a number of projects we have built and evolved fault-tolerant control systems. During this effort we have used both styles of development. We will now compare

	Manual FT	Meta-level FT
Error detection	Custom	Completely transparent
Fault repair	Custom	Promotes transparency
Error propagation	Exceptions	Exceptions
Consistency	Custom	General assumptions
Evolutionary pressure	Dependant	Low

Table 1 – Comparison of fault-tolerance approaches.

the two situations by discussing a number of topics. A summary of the comparison is given in table 1.

When using a manual approach, error detection can be customized when programming the fault container. This approach provides maximum flexibility. It also makes it possible to write multiple fault containers for the same service-provider interface. The main disadvantages of this approach are the burden of writing repetitive error-detection code, and the risk that it will be subject to evolutionary pressure.

Error detection using the meta-level approach is completely transparent. This is possible because error detection only relies on service-provider interfaces and not any client-specific notion of failure. This also means that only one error-detection behavior can exist for a given service-provider interface. When multiple slightly different error-detection requirements are needed, then programmers are forced to express these differences by declaring multiple service-provider interfaces.

The pros and cons of manual fault repair are similar to those of manual error detection. On the positive side, maximum flexibility is available. On the negative side, there is an extra programming burden, and a risk that custom code is subject to evolutionary pressure.

When using the meta-level approach, transparent fault repair is promoted to the greatest possible extent. Unfortunately, complete transparency is not possible. This is because the preferred order in which to try service providers is inherently domain specific. The order can be configured using a content-provider object. Note that content-provider objects do not per se violate transparency with respect to neither client components nor provider components. It is possible for a high-level component to configure the content-provider object a fault-tolerant lookup service should use. Thus, fault repair can always be made transparent with respect to client components and provider components, even though complete transparency with respect to the entire system is impossible.

With respect to error propagation there is usually no difference between a manual implementation and the meta-level approach. A manual implementation is almost certain to use exceptions. The meta-level approach does the same. As discussed in section 2, the main problems with state-of-the-art exception models are related to resumption of control flow, i.e. fault repair, and not error propagation.

When using a manual approach, preservation of consistency during fault repair can be achieved in a number of different ways. This choice influences what assumptions service-provider interfaces must adhere to. As discussed in section 4.5, preservation of consistency is particularly problematic when there is state to preserve.

In the meta-level approach, preservation of consistency is designed not to require participation by neither client components nor provider components. This is done by establishing a general set of assumptions for service-provider interfaces. Specifically, it is assumed that method invocations are atomic, provider objects are shared among multiple clients, and no provider-object state is transferred during fault repair. When state is unavoidable we advocate externalizing and sharing it.

A system's ability to evolve is dependent on the considerations discussed above. When fault-tolerance concerns such as error detection, fault repair, error propagation, and consistency preservation are transparent, then client components and provider components will not be subject to evolutionary pressure due to these concerns.

In our experience, it is difficult to manually implement fault tolerance, so that high transparency and low evolutionary pressure is achieved. When insisting on doing so, the solution converges towards a design with many similarities to the meta-level approach we have presented. Specifically, the design must achieve transparent fault tolerance and transparent sharing as discussed in section 3. To achieve transparent fault tolerance, fault containment can only depend on a service-provider interface and not on any specific service providers. To achieve transparent sharing, a lookup service can allow client components and provider components to enter and leave the system without generating evolutionary pressure.

In the meta-level approach, the above design decisions have already been taken. Thus, it is easier for the programmer to maintain a low evolutionary pressure in the presence of evolving fault-tolerance concerns.

Finally, it is our experience that meta-level fault tolerance is compatible with the component model promoted by the NetBeans Rich Client Platform. In practice, this means that we can introduce our new lookup service and apply it without deviating significantly from component-composition patterns and general practices used with NetBeans.

7 Conclusion

As argued previously, evolutionary pressure on manually programmed fault containers restrains evolution of both clients and service providers in component-based systems with fault-tolerance requirements. The source of the problem is lack of transparent fault tolerance and lack of transparent sharing.

We have proposed an approach, which provides the required transparency, by moving fault-tolerance concerns into a meta-level. The meta-level provides clients with dynamic fault containers created as a part of reference resolution at runtime. The behavior of a dynamic fault container depends solely on a service-provider interface and a (possibly domain-specific) content-provider object supplied by a high-level component.

We have applied our ideas to the NetBeans Rich Client Platform. This has been done by implementing a new lookup service and a dynamic fault container. In our experience the maintainability of fault-tolerance concerns during software evolution is improved. We also find that our approach is compatible with de facto development practice on the NetBeans Rich Client Platform. We believe that our ideas can be applied to any component model that supports lookup-based object wiring.

Reuse and evolution of independently developed components is a challenge in component-based systems with high requirements for fault tolerance. We believe that our meta-level-based fault-tolerance approach is a step towards improving the situation.

References

[AB87] A. Avizienis and D. E. Ball. On the achievement of a highly dependable and fault-tolerant air traffic control system. *Computer*, pages 84–90, 1987.

[AGH05] K. Arnold, J. Gosling, and D. Holmes. *Java Programming Language.* Addison-Wesley Professional, 2005.

[ALRL04] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, pages 11–33, 2004.

[BJPW99] A. Beugnard, J. M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[BTW07] T. Boudreau, J. Tulach, and G. Wielenga. *Rich Client Programming.* Prentice Hall, 2007.

[CA78] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. *Fault Tolerant Computing*, 1978.

[FN] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault-tolerant corba systems. *IEEE Transactions on Computers*, 53(5):497–511.

[GBR99] A. F. Garcia, D. M. Beder, and C. M. F. Rubira. An exception handling mechanism for developing dependable object-oriented software based on a meta-level approach. *Proceedings of Software Reliability Engineering*, 1999.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley Professional, 1995.

[Goe06] B. Goetz. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.

[Gra85] J. Gray. Why do computers stop and what can be done about it? *Technical Report 85.7*, 1985.

[GRRX01] A. F. Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.

[GS02] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. *Proceedings of the First Workshop on Self-Healing Systems*, 2002.

[HW03] G. Hohpe and B. Woolf. *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, 2003.

[jav] Class proxy. *Java Platform Standard Ed. 6.*

[KJ01] M. Kircher and P. Jain. Lookup. *Proceedings of 5th European Conference on Pattern Languages of Programs*, 2001.

[KJ04] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture*, volume 3. Wiley, 2004.

[KM85] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, 1985.

[KM90] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16:1293–1306, 1990.

[KMSL83] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *IEE Proceeding, Part E 130, 1*, pages 1–10, 1983.

[Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68:1060–1076, 1980.

[McG05] J. D. McGregor. Context. *Journal of Object Technology*, 4(7), 2005.

[Ran75] B. Randell. System structure for software fault tolerance. *ACM SIGPLAN Notices*, 1975.

[RX94] B. Randell and J. Xu. The evolution of the recovery block concept. *Software Fault Tolerance*, 1994.

[Sar03] T. Saridakis. Design patterns for fault containment. *Proceedings of the 8th European Conference on Pattern Languages of Programs*, pages 493–519, 2003.

[Szy02] C. Szyperski. *Component Software*. Addison-Wesley Professional, 2002.

[TH00] D. Thomas and A. Hunt. *Programming Ruby: A Pragmatic Programmer's Guide*. Addison-Wesley Professional, 2000.

[vN56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata Studies*, 1956.

[WS96] I. Warren and I. Sommerville. A model for dynamic configuration which preserves application integrity. *Configurable Distributed Systems*, 1996.

[XRRCS94] J. Xu, B. Randell, C. M. F. Rubira-Calsavara, and R. J. Stroud. Toward an object-oriented approach to software fault tolerance. *Fault-Tolerant Parallel and Distributed Systems*, 1994.

[XRZ] J. Xu, B. Randell, and A. Zorzo. Implementing software fault tolerance in c++ and openc++: an object-oriented and reflective approach. *Proceedings of International Workshop on Computer Aided Design, Test and Evaluation for Dependability*, pages 224–229.

[ZHR$^+$06] S. Zakhour, S. Hommel, J. Royal, I. Rabinovitch, T. Risser, and M. Hoeber. *The Java Tutorial: A Short Course on the Basics.* Prentice Hall PTR, 2006.

[Zim96] C. Zimmermann. Metalevels, mops and what the fuzz is all about. *Advances in Object-Oriented Metalevel Architectures and Reflection*, 1996.

About the authors

Martin Rytter is a PhD student at the Maersk Mc-Kinney Moller Institute, University of Southern Denmark. He can be reached at mlrj@mmmi.sdu.dk.

Bo Nørregaard Jørgensen is an associate professor at the Maersk Mc-Kinney Moller Institute, University of Southern Denmark. He can be reached at bnj@mmmi.sdu.dk.