Simplifying the Development of Autonomic Enterprise Java Bean Applications via Model Driven Development

Jules White, Douglas Schmidt, Aniruddha Gokhale
{jules, schmidt, gokhale}@dre.vanderbilt.edu
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville

Abstract

Autonomic computer systems aim to reduce the configuration, operational, and maintenance costs of distributed enterprise applications. In theory, autonomic systems can minimize the impact of human error in system development and management. In practice, however, it is hard to develop the monitoring, analysis, planning, and execution aspects required for highly complex autonomic software systems reliably within short timeframes. This paper provides two contributions to the development of autonomic computing systems using Enterprise Java Beans (EJBs). First, we describe the structure and functionality of a model-driven development (MDD) tool that can formally capture the design of EJB systems, their quality of service (QoS) requirements, and the autonomic properties that will be applied to the EJBs to support the rapid development of autonomic EJB applications via code generation, automatic checking of model correctness, and visualization of complex QoS and autonomic properties. Second, the paper describes how MDD tools can generate code to plug EJBs into a Java component framework that provides an autonomic structure to monitor, configure, and execute EJBs and their adaptation strategies at run-time. We present a case study that shows how these tools and frameworks help manage autonomic distributed applications at the EJB level, thereby allowing developers to incorporate adaptive properties at a finer granularity than simply restarting or cloning entire EJB applications.

Keywords: Autonomic Computing, Component Middleware, Quality of Service (QoS), Model Driven Development (MDD), and Enterprise Java Beans (EJBs).

1. Introduction

1.1 Autonomic Computing Challenges

Developing and maintaining distributed enterprise systems is hard, in part due to their complexity and the impact of human operator error, which studies have shown to be a significant contributor to distributed system repair and down time [2]. The goal of autonomic computing is to create distributed systems that have the ability to self-manage, self-heal, self-optimize, self-configure, and self-protect [1], thereby reducing human interaction with the system to minimize down-time from operator error. Although the benefits of autonomic computing are significant [1], the pressures of limited development timeframes and inherent/accidental complexities of large-scale software development have discouraged the integration of sophisticated autonomic computing functionality into distributed applications. Some enterprise application platforms, such as Enterprise Java Bean (EJB) [3] application servers, offer limited autonomic features, such as clustering, though they tend to have large development teams and long development cycles.

A key challenge limiting the incorporation of autonomic features into enterprise applications is the lack of design tools and frameworks for autonomic distributed systems that can alleviate complexity and generate code that is correct by construction. Some infrastructure does exist, such as IBM's Autonomic Computing Toolkit [4] that focuses on system-level logging and management. System-level autonomic toolkits are inadequate, however, for creating fine-grained autonomic capabilities.

To address the limitations with system-level autonomic toolkits, *component-level* autonomic frameworks are needed to help alleviate the development effort of creating autonomic applications. Component-level autonomic properties support more fine-grained healing, optimization, configuration, monitoring, and protection than system-level toolkits. For example, a mission-critical command and control system for emergency responders should be able to shutdown application component logic selectively as it fails, rather than shutdown the entire application. With component technologies, valuable functionality could still be maintained in this scenario. With existing autonomic infrastructure based on the system-level, the failure of a key component would trigger a restart of the entire application [5]. In contrast, a component-level autonomic framework could provide a mechanism to restart only the point of failure.

Creating applications with either system or component-level autonomic frameworks requires moving large amounts of state data, analysis data, actions plans, and execution commands between components. These types of applications also require careful weaving of monitoring, analysis, planning, and execution logic into the functional components of the system. Analysis of the autonomic aspects of the application, such as checking that the right state is being monitored by the right components, is a tedious and error-prone process. *Model driven development* (MDD) [25] is a promising means of reducing the cost associated with these activities. Models of autonomic systems can be constructed and checked

for correctness to ensure that system designs meet autonomic requirements. Tools can then be used to generate the various capabilities to move data, coordinate actions, and perform other autonomic functions.

1.2 Simplifying Autonomic System Design and Implementation via Model Driven Development

To address the need for a component-level autonomic computing framework, and to reduce the difficulty of *ad hoc* techniques that manually imbue autonomic qualities into distributed applications, we have created the *J3 Process*, which is a tool suite for the rapid design and implementation of autonomic applications. J3 consists of the following open-source[1] MDD tools and autonomic computing frameworks:

- **J2EEML**, which is a domain-specific modeling language (DSML) [13] that formally captures the design of EJB systems, their quality of service (QoS) [6] requirements, and the autonomic adaptation strategies of their EJBs. J2EEML constraint checkers help ensure that autonomic applications are constructed correctly and its models capture autonomic properties and reduce the design and implementation complexity of autonomic systems.
- **Jadapt**, which is a J2EEML model interpreter that generates code to plug EJBs into JFense, a component-level autonomic computing framework for EJB systems. The QoS and autonomic properties modeled via J2EEML can be analyzed by Jadapt. Jadapt in turn generates the EJBs and the glue code required to integrate them into **JFense**, which is a framework for monitoring, configuring, and resetting individual EJBs from an autonomic manager [8].

This paper provides several contributions to the development of autonomic computing systems using EJBs. First, we describe the structure and functionality of J2EEML and show how it simplifies the design of autonomic systems and facilitates the resolution of common autonomic system design challenges by providing notations and abstractions that are aligned with autonomic, QoS, and EJB terminology, rather than low-level features of operating systems, middleware platforms, and third-generation programming languages. Second, we describe how Jadapt generates EJB and Java code from J2EEML models to ensure that an autonomic application meets its specifications and to reduce implementation time. Third, we show how JFense provides a set of reusable autonomic components that allow developers to plug-in EJB applications and focus on autonomic logic, rather than the glue for constructing the autonomic system. Finally, we illustrate how the J3 Process significantly reduces the complexity of designing and implementing an autonomic EJB application.

[1] The J3 Process languages, tools, and frameworks are available from www. sourceforge.net/projects/j2eeml.

1.3 Paper Organization

The remainder of this paper is organized as follows: Section 2 describes the design challenges of building component-level autonomic EJB applications; Section 3 gives an overview of J2EEML, Jadapt, and JFense in the context of a representative distributed EJB application we developed as a case study; Section 4 compares our work with related research; and Section 5 presents concluding remarks.

2 Key Challenges of Developing Autonomic EJB Component Frameworks

Autonomic applications require four elements to achieve their goals: *monitoring, analysis, planning, and execution* [1]. These elements form a *controller* that attempts to monitor and adapt the application to maintain its goals. This section describes how each of these autonomic system elements should be designed to function properly and how applying MDD tools and frameworks helps simplify this process in the context of an EJB-based constraint-optimization system that schedules highway freight shipments using a multi-layered autonomic architecture, as shown in Figure 1.

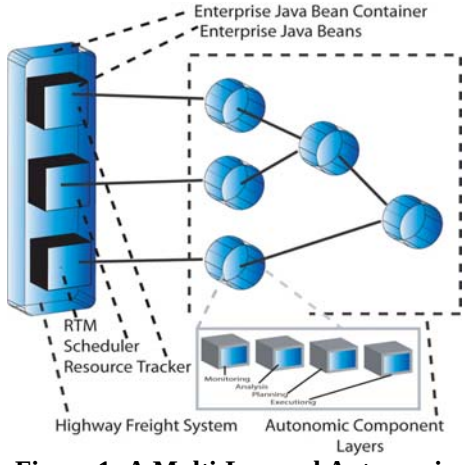

Figure 1: A Multi-Layered Autonomic Architecture.

Figure 1 shows the container and autonomic layers monitoring our example EJB application. Each autonomic layer contains monitoring, analysis, planning, and execution components. The highway freight scheduling system, used for our case study, is designed to assign drivers and trucks to freight shipments. The system has a list of freight shipments, which it must schedule. The system uses a constraints optimization engine to find a cost effective assignment of drivers and trucks to shipments. A central component to this system is the module for determining the route time from a truck's current location to a shipment start or end point. The *Route Time Module* (RTM) uses a geo-database and the GPS coordinates from the truck to perform the calculation. This

module will be the central part of our case study since it is crucial to the proper operation of the optimization engine. A heavy load is placed on the RTM and thus it is crucial that it maintains its QoS goals. The remainder of this section uses the RTM to illustrate key design challenges associated with autonomic systems. Section 3 then describes the J3 Process, MDD tools, and autonomic component frameworks we developed to implement the RTM and resolve these challenges.

2.1 Monitoring

In autonomic systems, monitoring is the means by which applications observe their own state. This state information becomes vital data that autonomic applications use to reason about how to manage themselves. It is therefore crucial that the right information be collected at the right times and in a manner that does not adversely impact the functionality and QoS of the system. The following are key design challenges faced when developing the monitoring aspects of an autonomic system:

1. Deciding what state information should be monitored
2. Deciding where the monitoring logic should reside in the autonomic application and
3. Deciding how and when to monitor.

We describe each of these challenges below and explain how they arise in the context of our highway freight scheduling system.

Challenge 2.1.1: Deciding what state information to monitor. The first design challenge developers of autonomic systems must address is what state information applications should self-monitor. For some systems, this information could include CPU and memory utilization. For other systems, exceptions thrown by the application or error messages in a log might be more relevant.

For example, one bottleneck in our highway freight scheduling system is the Route Time Module (RTM), which calculates the time required for a truck to reach a pickup location from its current position. This information is crucial to determine fuel costs and arrival estimates. If the RTM cannot keep up with demand, delivery quality will degrade since schedules will lag behind the actual positions of the trucks. If an empty truck is en route to a pickup and a new delivery request arrives, it might be more cost effective to divert the empty truck to service the new request and assign a different truck to handle its current request. This diversion is only feasible, however, if the current truck positions and route times are accurate. If the truck has already arrived at the original pickup location, but the scheduler is using earlier data where the truck has yet to arrive, it might redirect the truck to the new pickup, even though it is not cost-effective to redirect a truck that is picking up a shipment. The data provided by the RTM must not lag too far behind real-time, therefore, so the system needs to monitor the time needed to fulfill each route time request and adjust its routing calculation policies if it falls behind.

Challenge 2.1.2: Deciding where the monitoring logic should reside. The decision of what to monitor directly affects where the monitoring logic will reside. To monitor a log for errors, the logic could be at any level of the application, such as a central control level. For observing exceptions or the load on a specific subcomponent of the application, the monitoring logic must be embedded more deeply. In particular, developers must position the monitoring precisely so that it is close enough to capture the needed information, but not so deeply entangled in the application logic that it adversely affects application design that is tuned for performance and separation of concerns.

In our freight scheduling system, we must ensure separation of concerns in the application design and find an efficient means of monitoring. The monitoring logic for the RTM, however, should not be entangled with the route time calculation logic. Moreover, the time to monitor each request should be insignificant compared to the time to fulfill each route request.

Challenge 2.1.3: Deciding how and when to monitor. Another key design challenge that must be addressed is how to monitor, i.e., whether to use an active polling approach vs. a more passive event-based approach. Polling works well in some situations, but can waste resources by "busy waiting" when nothing is happening. Determining when to poll can also be challenging, i.e., polling too often can degrade performance, whereas polling too little can miss important state information. Within an EJB application, polling is even harder since the beans should not start or manipulate threads. The inability to create a timing mechanism to drive the polling period, makes polling hard to implement in EJB applications.

An alternate approach for monitoring is to listen for events, where components notify certain registered observers when their state changes. Event-based monitoring in EJBs can be implemented by observing a system log, e.g., using IBM's Generic Log Adapter [11] and Common Base Event format [12]. Another event-based approach is to broadcast events from specific components within the application, which performs all monitoring via method calls rather than string analysis of a log. This method forces developers to think carefully about the granularity of events to broadcast.

Event monitoring captures state information when interesting things occur, which can simplify the analysis logic since less superfluous data is examined and the problems associated with polling intervals are avoided. It is straightforward to implement event monitoring in an EJB application since EJBs do not impose any restrictions that make this strategy hard. The downside to event monitoring, however, is that the components must have the logic to determine when to broadcast events. For checking the status of a database connection, therefore, polling might be a better solution. With an event-based approach a component would only discover a broken connection when it attempted to use it, whereas a polling-based solution could discover the problem earlier.

In the context of our freight scheduling system, event-based monitoring makes more sense since the system is only concerned about the time for requests. An event-based system could easily notify the monitor of each request servicing time. A polling approach would require queueing servicing times so that information was not lost during polling periods. Queueing can introduce unneeded complexity, however, and might also create a significant lag time before a QoS failure was detected.

2.2 Analysis

In autonomic systems, analysis is the process of taking state information acquired by monitoring and reasoning about whether certain conditions have been met. For example, analysis often tries to determine if an application is maintaining its QoS requirements. The analysis aspects of an autonomic system can be centralized and executed on the totality of the system state or can be distributed and concerned with small discrete sets of the state. The following are key design challenges faced when developing an autonomic analysis engine:

1. Deciding what type of analysis engine (i.e., distributed vs. monolithic) best suits the application
2. Deciding how the engine should be decomposed and/or distributed
3. Designing the mechanism to move the data from the monitoring logic to the analysis engine.

We describe each of these challenges below and explain how they arise in the context of our highway freight scheduling system.

Challenge 2.2.1: Deciding what type of analysis engine suits the application. To choose a distributed vs. monolithic analysis engine the tradeoffs of each must be well understood. The concentration of the analysis logic into a single monolithic engine enables more complex calculations. However, for simple calculations, such as the average response time of the RTM component, a monolithic engine requires more overhead to store and retrieve state information for individual components than an analysis engine dedicated to a single component. A single analysis engine also provides a central point of failure. A key design question is therefore where analysis should be done and at what granularity.

The distinction between analysis and monitoring is not clearly defined. To understand the complex relationships between analysis and monitoring, consider the monitoring logic for our RTM. One way to build this logic would be to have a single monolithic analysis engine and monitor. The monitor would report the time to service each request back to the analysis engine, which would track the average request time and ensure it was below a certain threshold. A second option would be to only report request times above a threshold to the analysis engine and let it determine if too many requests were exceeding the threshold. In the first case, there is a clear separation between analysis and monitoring. In the second case, part of the analysis is done by the monitor to determine if the request time exceeds the threshold.

Challenge 2.2.2: Deciding how the engine should be decomposed and/or distributed. To create an effective analysis engine for our example application, developers must determine the appropriate number of layers. A key issue to consider is whether an application should have a single-layered vs. multi-layered analysis engine. At each layer, the original monitoring design questions are applicable, i.e., what should be monitored and how should it be monitored?

In the context of the RTM, a key question is whether its autonomic layer analyzes the RTM's response time or should a layer above the RTM do it. At each layer, the analysis design considerations are important too, e.g., what information the system is looking for in the data, how it finds this information, and how this can be better accomplished by splitting the layer. For example, a developer must consider whether every request to the RTM should be monitored to determine if the RTM is meeting its minimum response time QoS. Conversely, perhaps only certain types of requests that are known to be more time consuming should be monitored. Another question facing developers is how the RTM's monitoring logic sends data to its analysis engine.

Challenge 2.2.3: Designing the mechanism to move the data from the monitoring logic to the analysis engine. Analysis only works if the data can be moved from the monitoring logic to the analysis logic. The mechanism to perform this function must be crafted to handle both the type and volume of the information. It is also preferable for the mechanism's interface to be standardized and reused throughout the autonomic layers in the application. Creating a mechanism that meets all of these requirements is hard.

2.3 Planning

Planning is the phase in autonomic systems where applications examine the results of their analysis and decide what actions to take to reach their goals. For our highway freight scheduling example, this could involve changing the RTM to use a less precise but faster algorithm that maintains the minimum response time as demand grows. A typical autonomic application may have hundreds of goals and planning the correct actions in the face of QoS failures is critical to an autonomic application. The planning aspect of autonomic systems present the following key design challenges:

1. Deciding whether to use a single layer of planning or a multi-layer solution
2. Determining the responsibilities of each layer and
3. Deciding how each layer interacts with its surrounding layers.

We describe each of these challenges below and explain how they arise in the context of our highway freight scheduling system.

Challenge 2.3.1: Deciding whether to use a single layer of planning or a multi-layer solution. As with monitoring and analysis, planning can be implemented with a layered architecture. A simple, one-layer archi-

tecture would monitor, reason, and react to all system events at one level, which works well for macro-level events and actions. This simple approach is less feasible for applications that need more flexible and fine-grained the control of their behavior. To increase flexibility and fine-grained control, therefore, more layers can be integrated into the system. Layers distribute intelligence throughout the system and support a divide and conquer approach to planning.

Challenge 2.3.2: Determining the responsibilities of each layer. After the planning is provisioned into layers, each layer must be assigned a responsibility to react to and recover from QoS failures. In our highway freight scheduling example, one layer might ensure that the RTM is always available and the next layer down might ensure that a minimum response time is maintained. Intelligent separation of responsibilities can produce hierarchical chains of command that reduce the complexity of accomplishing the overall goal. Finding these well-proportioned divisions of labor is hard.

Challenge 2.3.3: Deciding how each layer interacts with its surrounding layers. Coordinating and integrating multiple layers into an application can be hard since each layer must interact with its neighboring layers. For example, restarting a subsystem might involve one layer issuing commands to restart each layer beneath it. Another issue is making the layers above aware of the plans that are about to execute. For example, if the layer that maintains minimum response time plans an adjustment to the RTM that will make it temporarily unavailable, the layer responsible for ensuring that the RTM is always available must be notified. The availability layer might also require veto power of the plans of the minimum response time layer to ensure that in-process transactions are completed before the changes proposed by the response time layer go into effect.

2.4 Execution

After an autonomic system has planned its course of action, it must execute its plans. Plans can range from restarting an entire application to applying configuration changes in a single component. Developers of autonomic systems need to address the following key challenges when designing their plan execution infrastructure:

1. How to send execution orders to the logic responsible for carrying them out.
2. How to carry out the actual execution and
3. How to coordinate actions that require multiple components.

To illustrate these challenges in the context of our highway freight scheduling example, we describe an autonomic layer that progressively restarts larger and larger portions of the RTM to recover from errors. This layer must first identify, analyze and plan which portion of the system the restart will initiate from. From that point, the autonomic layer must signal the faulty component to restart. After the restart is complete, the component must be further monitored and analyzed to determine if the error condition has been eliminated. If it has, the autonomic layer is done; otherwise the layer must attempt to restart a larger portion of the application.

For example, assume that the RTM depends on a single component to supply it with the current position of each truck and the current list of shipments that need servicing. We will refer to this component as the geo-locator component. If errors prevent this component from obtaining location information for the requests, it might try to restart a sub-component responsible for querying the truck location geo-database. If the restart of the truck location component fixes the problem, the system would cease restarting components. If the problem was not fixed, however, the component might try to restart both its truck location and shipment location components to alleviate the problem. At some point, if the problem could not be fixed, the geo-locator component might need to restart the entire system.

Challenge 2.4.1: How to send execution orders to the logic responsible for carrying them out. This design challenge involves determining how the autonomic system communicates with the original point of failure and coordinates the restarts. In a single layered system, this coordination would require the monolithic controller to directly access the geo-locator and its children. This access creates tight coupling of the controller to the implementation details of the geo-locator. An alternate approach is a distributed mechanism by which a central controller can delegate the restart to a component that has access to the component that needs to be restarted. Implementing the restart through delegation would be a layered approach to execution. Each layer would know how to restart larger and larger pieces of the system. The RTM could restart the coordinates system, the coordinates system could restart its children, and each child could restart its internal pieces.

A single monolithic execution controller, however, is not the most elegant approach to execution. By combining a layered approach to monitoring and analysis with layered execution, a system can be created in which levels higher in the hierarchy are only notified of error conditions if the current level cannot recover on its own. Each level monitors, analyzes, plans, and executes to achieve its own goals. If it cannot reach its goals, it notifies the layer above. This design places the recovery logic as close to the original point of failure as possible and also allows quicker detection and correction of errors. Another benefit is that it provides layers of abstraction that help to simplify the logic for each successive block of monitoring, analysis, planning, and execution.

Challenge 2.4.2: How to carry out execution orders. After a command has reached the autonomic layer responsible for performing it, the layer must be able to execute the order. Developers must therefore create an infrastructure to execute orders at runtime and allow those orders to change functional components.

Changing functional components requires that either the components themselves know how to execute the

orders or that they provide interfaces by which external agents can modify their operation. To provide reusable actions, the interfaces provided by the components must be generic enough that they can apply across the functional boundaries of the components. For example, the RTM and the pickup location system might both support a "restartable" interface that would allow the construction of a generic `restart()` operation, which could be applied to both components. To provide a flexible and reusable set of adaptive actions, developers must carefully understand the adaptation requirements of the system and how they can be factored into concerns that crosscut component boundaries. Obtaining a clear picture of the overall adaptive properties of the system and finding these concerns is hard.

Challenge 2.4.3: How to coordinate actions that require or affect multiple components. In a multi-layered system, executing an adaptation order may affect the components at more than one layer of the system. These multi-component effects can take two forms: (1) the action requires multiple components to adapt or (2) the adaptation of one component triggers changes in other components. In cases that require multiple components to adapt, developers must create a system to both synchronize and signal groups of components working together, which presents identical problems to those of distributed systems that require coordinated actions. For actions in one component that trigger changes in another, developers must clearly understand what these changes are and how the ripple effects will change the system as a whole. Careful attention must therefore be paid to ensure that seemingly innocuous adaptations do not produce unexpected and unwanted results elsewhere.

3 Using the J3 Process to Simplify Autonomic System Development

The J3 Process addresses the challenges of developing autonomic EJB applications discussed in Section 2. This process consists of the following steps:

1. The first component used in the J3 Process provides J2EEML, which is a DSML and MDD tool specifically tailored for autonomic applications. The key aspect of J2EEML that addresses many of the design challenges in Section 2 is the formal mapping from QoS requirements to application components.
2. The second component used in the J3 Process is a tool that produces many artifacts required to implement autonomic EJB applications modeled in J2EEML. This tool generates code that meets the J2EEML specifications and also reduces the amount of code that developers must write.
3. The third component used in the J3 Process is the JFense autonomic framework. JFense provides a standard set of components for monitoring, analysis, planning, and execution. These components can be used by developers to avoid writing custom autonomic frameworks. JFense is flexible and can be configured to meet the autonomic requirements of a wide range of EJB applications. It addresses the implementation related challenges, such as providing mechanisms to coordinate actions between layers and execute orders, challenges 2.4.2 and 2.4.3.

This section describes each component of the J3 Process and shows how they address key challenges of developing autonomic systems described in Section 2.

3.1 Resolving the Challenges of Autonomic Systems with J2EEML

To address the design challenges discussed in Section 2 we have developed J2EEML, which is a DSML for designing autonomic EJB systems that uses visual representations to model domain-specific abstractions, thereby enabling developers to construct models that incorporate autonomic and QoS concepts as first-class entities. The aim of J2EEML is to formally capture the relationship between QoS goals and application components. Capturing this relationship allows developers to address many of the design challenges from Section 2.

For example, developers can clearly understand which components to monitor in the application since they can visualize the relationships between components and QoS goals. This understanding facilitates intelligent decisions about what to monitor (challenge 2.1.1), and where monitoring logic should reside (challenge 2.1.2).

Developers can also design hierarchical QoS goals to divide and conquer complex QoS analyses. Modeling QoS goals hierarchically provides the following:

1. The ability to understand what type of analysis engine to choose (challenge 2.2.1). A small number of complex QoS goals that cannot be broken into smaller pieces imply the need for a monolithic analysis engine. A large number of goals – especially hierarchical QoS goals – imply the need for a multi-layered analysis engine.
2. The ability to understand how to decompose the analysis engine into layers (challenge 2.2.2). The hierarchical model of the QoS goals directly corresponds to the decomposition of the analysis engine in layers. A developer first adds a complex QoS goal to the model. The developer then determines if the complex goal can be accomplished by combining the results of several smaller analyses. If so, the developer adds these smaller QoS goals as children of the original QoS goal to represent the smaller analyses and then applies this iterative process to the new children.

Developers can also associate adaptation plans with each QoS goal to design the planning aspects of the autonomic application and aid in the following two tasks:

1. Choosing a single-layer or multi-layered planning architecture (challenge 2.3.1). If a complex QoS goal does not have adaptation plans associated with its children, it implies that the correct course of action to take when one of the child QoS goals fails cannot be de-

termined by the data available to the child. If only top-level QoS goals have associated adaptation plans, this implies the need for a single planning layer. If, however, the QoS children have adaptation plans associated with them, this implies that they can determine the corrective course of action and require a multi-layered planning solution.
2. The adaptation plans associated with a QoS goal indicate the responsibilities of that autonomic layer (challenge 2.3.2), i.e., the adaptation plan specifies the actions that the autonomic layer is responsible for choosing from in the event of a QoS failure.

The mapping between QoS goals and components facilitates the generation of code artifacts to reduce application development time. For example, understanding which components need to be monitored and analyzed allows external tools to generate proxies and interceptors for the monitored components. These interceptors can relay state information to the monitoring and analysis logic. Code can then be generated that moves the monitoring and analysis results to the appropriate planning layers. Other code generation abilities are facilitated and will be discussed in detail in Section 4.1.

J2EEML captures the design of an autonomic system and the mapping of components to QoS goals in the following four phases:
1. Developers create a structural model of the EJBs composing the system
2. Developers create models of the goals that the system is attempting to attain
3. The goals are mapped to the specific beans within the system that are responsible for them and
4. Developers create adaptation plans and associate them with QoS goals.

This modeling process captures the structure of the system, how the QoS properties are related to the structure, and what adaptation should take place if a QoS goal is not met. It is also well-suited for creating both the single and multi-layered models described in Section 2.3.

J2EEML was developed using the Generic Modeling Environment (GME) [14] created by the Institute for Software Integrated Systems (ISIS) at Vanderbilt University (see Sidebar 1 for an overview of GME). A GME-based metamodel describing the problem domain was constructed and interpreted to create the J2EEML DSML for autonomic EJB systems. J2EEML models constructed in the domain are interpreted by Jadapt, which is a tool for generating Java code, EJB skeletons, and necessary EJB deployment descriptors for JFense, which is a component-level autonomic framework for facilitating monitoring, analysis, planning, and execution in a Java application.

3.1.1 EJB Structural Model

The first component of a J2EEML model is its EJB structural model, which describes the components of the system that will be autonomically managed. This model defines the beans that compose the system and captures the EJB specifics of each bean, including JNDI names, transactional requirements, security requirements, package names, descriptions, remote and local interface composition, and bean-to-bean interactions.

Sidebar 1: Overview of GME

GME is an MDD environment that provides the following capabilities:
- A visual interface that allows building domain-specific modeling languages i.e., GME contains a metamodeling environment that supports the definition of paradigms, which are type systems that describe the roles and relationships in particular domains.
- Allows creation of models that are instances of these modeling language paradigms within the same environment.
- Customize such environments so that the elements of the modeling language represent the elements of the domain in a much more intuitive manner than is possible via third-generation programming languages.
- Allows building libraries of such environments, thereby supporting composition of modeling languages.
- Has a flexible type system that allows inheritance and instantiation of elements of the modeling languages.
- Provides an integrated constraint definition and enforcement module based on OMG's Object Constraint Language (OCL), which enables defining rules that must be adhered to by elements of the models built using a particular modeling language.
- Facilities to plug-in analysis and synthesis tools that operate on the models.

GME is available in binary and open-source form from www.isis.vanderbilt.edu/Projects/gme/.

The EJB structural model is constructed via the following steps:
1. Each session bean in the system is added to the model by dragging and dropping session bean atoms into the model. Designers then provide the Java Naming and Directory Interface (JNDI) name of the bean, its description, and its state type (stateful or stateless).
2. For each session bean, a model is constructed of the business methods and creators supported by the bean by dragging and dropping method and creator atoms. Each method and creator atom has properties for setting the visibility, whether it is part of the local interface, remote interface, or both, the transactional requirements, the security role requirements, and the input/output.
3. Entity beans are dragged and dropped into the model to construct the data access layer. These beans are provided a JNDI name/description and properties indicating if they use container managed persistence (CMP) or bean managed persistence (BMP).
4. Persistent fields, methods, and finders are dragged and dropped into the entity beans. Each persistent field has properties for setting visibility, type, whether it is part

of the primary key, and its access type (read-only or read-write).

5. Relationship roles are dragged and dropped into the entity beans and connected to persistent fields. These relationship roles can be connected to other relationship roles to indicate entity bean relationships.
6. Connections are made between beans to indicate bean-to-bean interactions. Capturing these interactions allows Jadapt to later generate the required JNDI lookup code for a bean to obtain a reference to another bean.

After these six steps have been completed, the J2EEML model has enough information to represent all the structural parts of the EJBs. The only structural aspect not modeled is the implementation of the logic for the business methods. Figure 3 shows a J2EEML structural model of the highway freight scheduling system described in in Section 2.

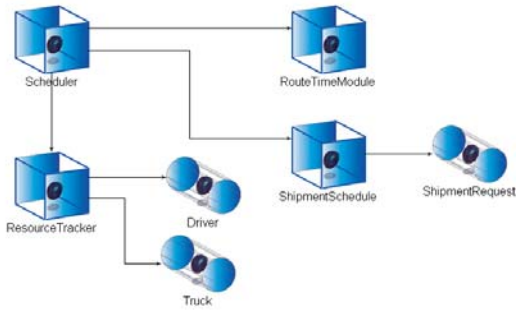

Figure 3: J2EEML Structural Model.[2]

To support decomposition of complex architectures into smaller pieces, J2EEML allows EJB structural models to contain child EJB models (solutions). Beans within the sub-solutions show up as ports that can receive connections from the parent solution. This design allows developers to decompose models into manageable pieces and enables different developers to isolate their designs.

For our highway freight scheduling example, we constructed a structural model of each bean required for the RTM, constraint-optimization engine, truck status system, and incoming pickup request system. This structural model can be seen in Figure 3. The model also included information on the entity beans used to access the truck location database and the pickup request database.

Using the J2EEML model based approach provided following advantages in the design phase:

1. Clear visualization of the beans and their interactions
2. Visualization of component security requirements
3. Visualization of system transactional requirements
4. Visualization of interactions between beans
5. Enforcement of certain EJB best practices, such as the Session Façade pattern [10], which hides Entity beans from clients through Session beans.
6. Model correctness checking, including checks for proper JNDI naming.

[2] All J2EEML screenshots have been cropped to focus on the models presented within them.

The visualization benefits significantly decreased the difficulty of understanding the system structure and interaction. The correctness checking and enforcement of best design practices facilitated rapid creation of both a correct by construction and well-designed solution.

3.1.2 Goal Modeling

The next component of a J2EEML model is the QoS goals model, which contains the properties that the autonomic aspects of the system are concerned about. The QoS goals determine what gets monitored and analyzed (Challenge 2.1.1). Goals are modeled in J2EEML as QoS properties that the system needs to maintain. Developers drag and drop goals into J2EEML models. Each goal provides properties for setting its name and description. Complex goals can be constructed hierarchically by dragging and dropping sub-goals into a parent goal. These sub-goals represent the QoS properties that must be maintained for the parent to succeed.

The QoS goals model is critical for understanding an autonomic system's QoS properties and choosing the appropriate monitoring, analysis, planning, and execution architecture. It is also crucial to designing the structural architecture of the EJB application and understanding how it meets those goals. Capturing and mapping QoS requirements to the appropriate structural architecture have traditionally used textual descriptions, such as "the service must support X users with an average access time of Y." Due to the lack of an unambiguous formal description, such descriptions are prone to different interpretations, which result in architectures that do not meet the QoS requirements. Choosing an EJB architecture that best fits the QoS requirements can be complex and error-prone since specification ambiguity and hidden trade-offs in architectures make it hard to choose the appropriate design.

For example, with a J2EE implementation of a service, deciding whether to use remote interfaces or not can have a substantial impact on end-to-end system QoS. Remote interfaces allow for distribution of beans across servers, which can increase scalability. Distribution can also increase latency, however, since requests must travel across a network or virtual machine boundaries.

Understanding the hierarchical relationships between QoS properties can be used not only to design the appropriate structural architecture, but also to determine the layering scheme used to monitor, analyze, plan, and execute (challenges 2.1.2, 2.2.1, 2.2.2, 2.3, 2.3.1, 2.3.2). If only a few large-scale QoS goals are required, a monolithic approach is appropriate. As the complexity of the relationships between QoS goals increases, multi-layered solutions become more appropriate. By visualizing the QoS properties and hierarchical relationships via J2EEML models, developers can better understand what approach(es) to apply.

3.1.3 Goal-to-EJB Mapping

After the structural and goal models are completed, developers can use J2EEML to create a mapping from the QoS goals to the EJBs in the structural model. This mapping documents what portions of the system are responsible for which QoS goals. It also indicates where monitoring, analysis, and adaptation need to occur for an autonomic system to maintain those goals (challenges 2.1.2 and 2.2.2).

Each QoS goal in J2EEML can be associated with multiple EJBs, which supports aspect [15] type modeling of QoS goals when they crosscut component boundaries. For example, maintaining a certain minimum response time might be crucial for several different components. Connecting multiple components to a QoS goal rather than creating a copy for each component produces clearer models. It also clearly shows the connections between components that share common QoS goals. Figure 4 shows a mapping from QoS goals to EJBs.

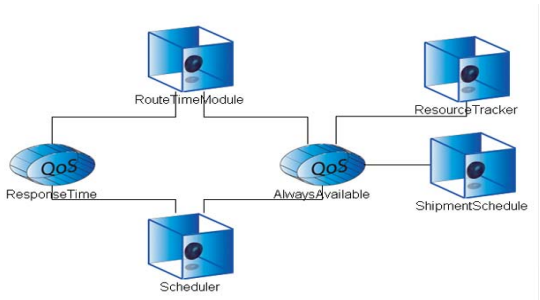

Figure 4: J2EEML Mapping of QoS Goals to EJBs.

Components can have multiple QoS goal associations, which are accomplished in J2EEML by either creating a single goal for the component that contains sub-goals or by connecting multiple QoS goals to the component. If the combination of goals produces a meaningful abstraction, hierarchical composition is preferred. For example, a QoS goal called "Always Produces Correct Result" could be constructed from the sub-goals "No Exceptions Thrown" and "Never Returns Null." Combining "Minimum Response Time" and "No Exceptions Thrown," however, is unlikely to produce a meaningful higher-level abstraction, so the multiple connection method is preferred in this case.

Once goals are created, each goal can have an associated adaptation plan to execute if the goal is not being met. Associating adaptation plans with strategies provides a way to formally capture the proper course of action in the event of a QoS failure. It also allows developers to create hierarchical adaptation plans.

3.2 Reducing the Complexity of Implementing an Autonomic System with Jadapt and JFense

JFense is a component-level autonomic framework that addresses many challenges described in Section 2 involved with developing autonomic systems. It provides a standard set of components and interfaces for monitoring the QoS of EJBs, analyzing system state, communicating between autonomic layers, determining how to adapt to QoS failures, and executing complex adaptation plans. Jadapt serves as the bridge between the J2EEML model (Section 3.1) and the JFense framework (Section 3.2.2). Jadapt generates Java code for the structural model represented in J2EEML and also generates glue code to plug the generated EJBs into the JFense framework. The goals of JFense and Jadapt are to:

- **Provide a standard framework for performing autonomic functions in an application**. JFense can be configured to mirror the autonomic architecture described by the J2EEML model. JFense is responsible for gluing the monitoring, analysis, planning, and execution logic of the application. JFense alleviates developers from reinventing an autonomic framework for their application. It also handles the communication between layers (challenge 2.2.3) since application developers do not have to write code to move the data from the monitoring logic to the analysis logic. JFense drives the communication between layers (challenges 2.3.3 and 2.4.1) since developers do not write the code to relay/coordinate planning information and adaptation orders. JFense provides a configurable event-based monitoring framework (challenge 2.1.3). Finally, JFense contains an adaptation system that can coordinate adaptation actions between layers and execute the actions (challenges 2.4.2 and 2.4.3).
- **Rapid development and verification of autonomic code through model interpretation**. Jadapt generates configurations for JFense to mirror the J2EEML model, stubs for the EJBs, EJB deployment descriptors, and monitoring, analysis, planning, and execution class stubs, which relieves the developer of a considerable set of coding tasks. Moreover, Jadapt ensures that the code mirrors the system architecture in J2EEML implementation, which reduces problems from misinterpretation of the specification and inconsistencies between implementations and interfaces.

The remainder of this section describes how J2EEML, Jadapt, and JFense collaborate to simplify the development of autonomic EJB applications.

3.2.1 Simplifying the Creation of Autonomic Applications that Meet Specifications with Jadapt

Jadapt is a J2EEML model interpreter that generates implementations of EJBs from a structural model defined using J2EEML. Figure 6 illustrates the Jadapt generated code. These classes include stubs for the methods and creators declared, as well as classes for plugging EJBs into JFense, which is our autonomic computing framework described in Section 3.2.2. Using the inter-bean interactions from the structural model, Jadapt generates code to get references to the homes of the EJBs

referred to by each bean, which frees developers from writing boilerplate code to obtain these references each time they need them. The design of the generated JNDI mechanism centralizes the lookups in one class, which makes it possible to implement caching solutions if needed. Generation of the JNDI lookup code also ensures that the correct JNDI names are used. Developers call get methods that return an instance of the home interface they need. Usually, discovering JNDI lookup errors cannot be done until run-time, which is far more costly and hard to debug.

Jadapt assumes that developers will modify the generated beans outside of the J2EEML development environment. Bean classes are therefore marked up with XDoclet attributes to automate the synchronization of the bean class, interfaces, and descriptors. XDoclet reads these attributes and generates the required interfaces and deployment descriptor XML. Developers only need to maintain the central bean class and synchronize the interfaces to it with XDoclet.

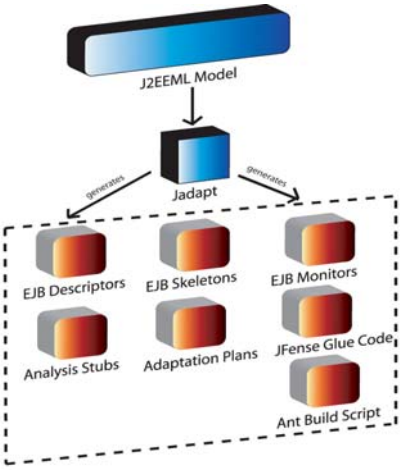

Figure 6: Jadapt Code Generation

The bean descriptor XML generated by Jadapt includes all of the transactional, security, visibility, relationship, and container type properties declared in the model. For example, if method `getCoordinates()` on class `RTM` is given a limited set of security roles that can access it, this security declaration will be included in the generated bean descriptor, which ensures that the design, deployment, and configuration are in sync. Generating the bean descriptor eliminates typographical errors from hand editing of descriptor XML.

For our highway freight scheduler, Jadapt reduced the development time of the application by generating:

1. All skeleton classes and interfaces required for EJBs
2. XDoclet attributes to maintain class, interface, and deployment descriptor synchronization during the development cycle
3. XML deployment descriptors for the EJBs
4. An Ant build infrastructure
5. Project documentation from descriptions captured in the model
6. Glue code to plug the generated EJBs into JFense

Since developers can easily adjust the application design and regenerate the application skeleton, their efforts are focused on designing the application and implementing the logic required for the business methods.

3.2.2 Simplifying Autonomic Application Development with the JFense Framework

To simplify the development of autonomic EJB applications, we created the JFense framework for constructing autonomic EJB systems. JFense provides a multi-layered architecture for monitoring, analyzing, planning, and executing in an autonomic system.

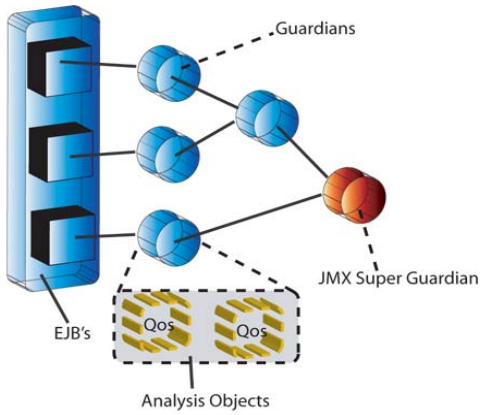

Figure 7: The JFense Multi-Layered Architecture.

The basic structure of JFense is shown in Figure 7 and is defined as follows:

1. Each bean has a guardian class responsible for monitoring its state and running QoS analysis. The beans push state data out to the guardians using an event-based system. The guardians act as observers on the beans [17], i.e., they are the key elements for monitoring beans and routing state information to the proper QoS analysis objects.
2. An analysis class for each QoS goal is created. These QoS goals are used by the guardians to analyze the bean's current state and determine if it is meeting its QoS requirements. Hierarchical QoS goals are created through aggregation.
3. Each guardian class has an associated action plan for determining the course of action if a QoS goal fails. The guardian also notifies any guardians at the level above when it cannot maintain its QoS goals.
4. A central Java Management Extension (JMX)-enabled bean resides at the top of the hierarchy and serves as a super guardian. This bean can be remotely managed and configured through JMX.

When a bean's state changes, it notifies its guardian that a state change event has occurred. The guardian then uses its QoS analysis objects to analyze the bean's state and ensure that its objectives are still being met.

Bean requests are the default state information monitored by guardians. Jadapt generates proxies that monitor

the input, output, time, and exceptions thrown for each method accessible through the beans local or remote interface and pass it to the Guardians. This coordination of proxies and guardians applies the Proxy [19, 20] and Interceptor [20] patterns, which intercept requests to an object before delegating the request to the actual object. This functionality can be adapted for unit testing before planning logic has been developed. In fact, logic for unit tests can be later recycled to form real-time QoS tests.

For example, a unit test may be constructed to ensure that a method of the RTM never throws an exception. It may not be feasible, however, to cover every possible input combination to a method during unit testing. Assume a method passes the unit test, but developers identify certain incorrect configurations under which the method could throw an exception. A QoS test can be constructed from the original unit test that monitors whether or not the method ever throws that exception. If that particular exception is thrown it indicates a misconfiguration, the guardian can then take steps to fix the bean's configuration.

Beans monitor requests on their accessible methods through generated proxies. When a request is issued to the bean, the generated proxy first receives the request and notes the start time. The proxy then notifies the guardian that a request is starting so that any pre-conditions on the request can be analyzed. These pre-conditions can be used to identify QoS failures in other portions of the system. The proxy then passes the request to the actual method that contains the logic to fulfill it (we refer to this method as the *implementing method*). When the implementing method has returned, the bean again notifies its guardian, which enables the guardian to check post-conditions, such as output correctness or servicing time. Finally, the result is passed back to the caller. This system of intercepting requests and passing the state events to the analysis objects relieves application developers from manually implementing state event and monitoring capabilities (challenge 2.2.3).

As described above, each request incurs monitoring overhead. This monitoring, however, need not happen on a per-request basis, i.e., the guardian can be tuned to only listen at certain event intervals or not to begin listening until activated by another portion of the system. The guardian can also begin monitoring when an exception is raised during a request. The guardians use the Strategy pattern [17] to allow the appropriate monitoring scheme to be configured (challenge 2.1.4).

After the state is routed to the analysis object, it determines if its QoS property is being met. JFense has several predefined analysis objects for common functions, such as monitoring request time. Other autonomic analyses can be added by extending the JFense analysis interfaces or implementing the class skeletons generated by Jadapt from the J2EEML model. If the QoS is not being maintained, the analysis object notifies the guardian, which will either directly execute an action plan or propagate the QoS failure event up the chain of guardians. This chain of communication between guardians provides a solution to the problem of coordinating multiple layers (challenge 2.3.3).

Guardians also use the Strategy pattern to determine how to react to a QoS failure. Different planning strategies can be configured into a guardian at design- or run-time to find the appropriate course of action for each QoS failure. The default strategy uses a hashing scheme to associate QoS analysis objects with Command pattern actions, which encapsulates an action as an object, to allow requests to be queued, logged, or undone. In the event of a QoS failure, the appropriate action is looked up from the table and executed.

To provide a flexible means of controlling components, the guardians implement the Extension Interface pattern [24], which is to provide a mechanism to query a component for the interfaces it supports and obtain an interface. When an action is executed, it is passed a reference to the guardian that observed the QoS failure. The action can use the guardian to locate objects that support the interfaces it needs to accomplish its goals. In the case of the RTM, this might involve looking up a `Restarter` interface to allow the action to restart the RTM after an error occurred.

JFense alleviates developers of the need to build an autonomic framework from scratch for the highway freight scheduling system. JFense handles inter-layer communication so that developers can focus on the logic needed to analyze the state data, determine the correct course of action, and adapt the system. JFense also provides the communication, monitoring, and bus infrastructure to glue the provided logic together, which significantly reduces the time and effort required to build autonomic applications that monitor their own state and adapt to achieve their goals.

4 Related Work

This section reviews work on aspect-oriented programming, J2EE instrumentation, J2EE management, and middleware for adaptive QoS provisioning. We look at a group of technologies that are combined to form autonomic applications. These technologies are not combined by any research effort, however, into a single solution for autonomic computing.

Aspect-oriented programming. Separating concerns that crosscut multiple layers of an application is hard. Aspect-oriented programming (AOP) offers meta-programming mechanisms that enable developers to specify *aspects* or crosscutting concerns and inject them across multiple levels of an application. This injection can be done at compile-and/or run-time. Two such facilities for Java are AspectJ [21], which imbues objects with the meta-defined behaviors at compile-time, and AspectWerkz [22], which uses Java reflection to attach the behavior at run-time.

AspectJ and AspectWerkz have been used to monitor the performance of pre-existing J2EE applications.

These aspect-oriented languages help imbue autonomic capabilities into existing systems. They take the approach of modifying the application itself rather than the common approach of retrofitting the application's logging mechanism to fit Common Base Event (CBE) or other easily analyzed format.

When designing an autonomic application, the use of patterns, such as the Interceptor pattern [24] and the Smart Proxy pattern [19, 20], can minimize the need for an aspect-oriented approach. JFense uses the Proxy/Interceptor approach since it is not currently intended to retrofit existing applications.

AspectJ and AspectWerkz can be used to weave autonomic concerns into an application. They do not however provide any means of addressing the challenges discussed in sections 2.1–2.4. J3 provides a development solution that directly addresses the challenges in 2.1–2.4. AspectJ and AspectWerkz are a means to help integrate autonomic functions. The J3 process provides an autonomic development solution that covers application design to code generation to runtime monitoring, analysis, planning and execution.

J2EE instrumentation and management. Work on J2EE instrumentation is closely related to monitoring and out-of-band adaptation [18]. Java Management eXtensions (JMX) provide a standardized mechanism for remotely instrumenting and managing Java applications. It can be used to remotely coordinate the autonomic nervous systems of several applications. Although remotely accessible EJBs could perform the same function, JMX is designed specifically for management and instrumentation. Custom EJB solutions would inevitably reinvent much of JMX's functionality. A custom solution would also be incompatible with existing remote management consoles.

JMX aids in the development of autonomic systems, but as with aspect-oriented programming, does not by itself provide an autonomic framework. JFense uses JMX to offer inter-application coordination of autonomic capabilities. Again, J2EEML, Jadapt, and JFense provide end-to-end autonomic application development and not merely generic glue to attach autonomic capabilities.

Adaptive and reflective middleware. Adaptive and Reflective middleware analyzes its own state and adapts its QoS provisioning to maintain pre-defined levels of service. The dynamic provisioning of QoS properties in middleware mirrors the challenges faced in autonomic systems. Care must be taken to separate the concerns of monitoring and adaptation from application logic [18]. The Proxy [17] and Interceptor [24] patterns are familiar to CORBA applications. Qoskets [21] and Quality Objects (QuO) [23] are similar in intent to JFense, in that they focus on separating the monitoring of QoS from the application logic and provide introspection mechanisms by which applications can observe and react to changes in their own state. The entire J3 Process not only includes an autonomic framework but an end-to-end modeling and code-generation solution. J3 aids not only by providing a framework but by facilitating system design and specification. Jadapt is J3's mechanism for ensuring that the autonomic application meets the specification. Using a modeling to run-time autonomic solution, ensures that the application is properly designed, meets its goals, and is a proper implementation of the autonomic specification. Qoskets and QuO do not provide model verification or a mechanism to ensure that the application meets its specifications.

5 Concluding Remarks

Significant challenges arise when developing EJB applications with autonomic capabilities. In particular, developers must reason about complex sets of QoS goals and ensure that applications meet them. Autonomic capabilities provide a means for EJB applications to self-manage and attempt to maintain the QoS goals. To facilitate self-management, both the structure of EJB applications and their QoS goals must be captured formally so that applications can reason about themselves.

The bridge between the QoS goals of autonomic systems and their structural designs involves mapping these goals to specific system components. Without this mapping, applications cannot use introspection to determine whether their QoS goals are being met. The J2EEML MDD tool described in this paper helps link goals and structure by allowing developers to specify this mapping formally via a domain-specific modeling language. J2EEML also includes mechanisms for describing complex EJB structures, interactions, and architectures.

After the structural properties, QoS goals, and goal to structure mapping have been captured by J2EEML, developers still must integrate autonomic features into their distributed EJB applications. This integration is often overly complex due to the lack of component-level frameworks for application autonomicity. To address these concerns, we have developed the Jadapt code generation tool and the JFense autonomic framework. Jadapt allows developers to generate the required code to plug their application's EJBs into JFense. JFense provides a comprehensive and flexible framework for multi-layered autonomic monitoring, analysis, planning, and execution architectures, which allows developers to focus on the system's business logic and QoS analysis logic. The J3 Process provides a component-level toolkit that simplifies the development of autonomic EJB applications.

From the development of the J3 process, we have learned that creating reusable components to monitor a system and execute adaptation plans is a challenging endeavor. Not all applications will want to monitor the same types of data sets. The monitoring framework must therefore be flexible to incorporate unanticipated data sets, yet it also needs to handle the most common cases well. Striking this balance between flexibility and general case utility takes patience and iteration.

We have also learned that developing adaptations for an application is a tricky process. Most developers do

not think about designing components that can be adapted, swapped, restarted, or reconfigured to handle errors. Trying to build a component and separate the adaptive concerns is hard.

In future work, we are developing a series of increasingly sophisticated autonomic distributed applications using our MDD tools. These applications will serve as a testbed for investigating various autonomic architectures, monitoring strategies, and planning strategies. We are also enhancing our tools to increase their expressive and code generation capabilities. We plan to integrate our MDD tools with CIAO [25], which is a QoS-enabled component middleware framework that implements the CORBA Component Model (CCM).

References

[1] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. Computer, 2003.

[2] D. Oppenheimer, A. Ganapathi, D. Patterson, "USENIX Symposium on Internet Technologies and Systems," 2003.

[3] V. Matena, M. Hapner, "Enterprise Java Beans Specification, Version 1.1," Sun Microsystems, Dec. 1999.

[4] Autonomic Computing Toolkit, IBM, www106.ibm. com/developerworks/autonomic/overview.html.

[5] G. Candea, A. Fox, "Designing for High Availability and Measurability," Proceedings of the 1st Workshop on Evaluating and Architecting System Dependability (EASY), July 2001.

[6] R. Schantz, J. Loyall, D. Schmidt, C. Rodrigues, Y. Krishnamurthy, I. Pyarali, "Flexible and Adaptive QoS Control for Distributed Real-time and Embedded Middleware," in Proceedings of Middleware 2003, 4th International Conference on Distributed Systems.

[7] J. Sztipanovits, G. Karsai, "Model-Integrated Computing," IEEE Computer, April 1997.

[8] T. Eymann, M. Reinicke, et al., "Self-Organizing Resource Allocation for Autonomic Networks," DEXA Workshops, 2003.

[9] R. Johnson, "J2EE Design and Development," Wiley, 2003.

[10] D. Alur, J. Crupi, D. Malks, "J2EE Core Patterns," Sun Microsystems Press, 2003.

[11] E. Giguere, "Create GLA components using Release 2 of the Autonomic Computing Toolkit," IBM Developerworks, (www106.ibm.com/ developerworks/edu/ac-dw-ac-glacomp2i.html? S_TACT=104AHW20&S_CMP=HP).

[12] " Specification: Common Base Event ," IBMDeveloperworks, (www106.ibm.com/developerworks/ webservices/library/ws-cbe/).

[13] A. Ledeczi, A. Bakay, M. Maroti, P. Volgysei, G. Nordstrom, J. Sprinkle, and G. Karsai, "Composing Domain-Specific Design Environments," IEEE Computer, Nov. 2001.

[14] The Generic Modeling Environment, Institute for Software Integrated Systems, Vanderbilt University, (www.isis. vanderbilt.edu/Projects/gme/GME reference).

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," ECOOP Object-Oriented Programming, 11th European Conference, 1997

[16] XDoclet, (xdoclet.sourceforge.net/xdoclet/).

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software,"Addison-Wesley, 1995.

[18] N. Wang, C. Gill, D. Schmidt, A. Gokhale, et. al, "Middleware for Communications," John Wiley & Sons, Ltd, 2001

[19] R. Koster, T. Kramp, "Loadable Smart Proxies and Native-Code Shipping for CORBA," in *Proceedings of the Third IFIP/GI International Conference on Trends towards a Universal Service Market (USM), Munich*, pp. 202–213, September 2000.

[20]. N. Wang, K. Parameswaran, and D. Schmidt, "The Design and Performance of MetaProgramming Mechanism for Object Request Broker Middleware," in *Proceedings of the 6th USENIX Conference on Object- Oriented Technologies and Systems (COOTS'01)*, January 2001.

[21] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, "QoS Aspect Languages and Their Runtime Integration.," in *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components*, 1998

[22] AspectWerkz, (aspectwerkz.codehaus.org/index-aw.html)

[23] J. Zinky, D. Bakken, R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, 1997

[24] D. Schmidt, M. Stal, H. Rohnert, F. Buschmann, "Pattern-Oriented Software Architecture," John Wiley and Sons, Ltd., 2000.

[25] N. Wang, D. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. Loyall, R. Schantz, and C. Gill, "QoS-enabled Middleware," in *Middleware for Communications*, edited by Q. Mahmoud, Wiley and Sons, New York, 2003.

[26] Greenfield et al., "Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools," Wiley and Sons, 2004.