Flexible Self-Management Using the Model-View-Controller Pattern

Edward Curry, *National University of Ireland, Galway*

Paul Grace, *Lancaster University*

A self-management infrastructure requires a self-representation to model system functionality concerns. The Model-View-Controller design pattern can improve concern separation in a self-representation.

Future computing initiatives such as ubiquitous and pervasive computing, large-scale distribution, and on-demand computing will foster unpredictable and complex environments with challenging demands.[1,2] Next-generation systems will require flexible system infrastructures that can adapt to both dynamic changes in operational requirements and environmental conditions, while providing predictable behavior in areas such as throughput, scalability, dependability, and security. Successful projects, once deployed, will require skilled administration personnel to install, configure, maintain, and provide 24/7 support.

To meet these challenges head-on, computing systems will need to be more self-sufficient. IBM's vision of autonomic computing is an analogy with the human nervous system that coordinates low-level routine bodily functions such as respiration, muscle activity, and perspiration.[3] An autonomic, or self-management, computing system would relieve the burden of low-level functions such as installation, configuration, dependency management, performance optimization management, and routine maintenance from their conscious brain: the system administrators.

Self-management systems must be flexible and customizable. An important part of a self-management infrastructure is the self-representation used to model system functionality concerns, allowing runtime inspection and adaptation. As the range of self-management capabilities expands, the task of creating appropriate self-representations becomes ever more complex. Current design practices for self-representations are inflexible and therefore costly to change. Appropriate concern separation in a self-representation is vital. The Model-View-Controller (MVC) pattern can improve concern separation by helping encapsulate state, analysis, and realization operations. This in turn will improve the self-representation's flexibility and customization, while simplifying portability between system implementations. Here, we evaluate the merits of an MVC-based self-representation design and demonstrate its improvements in flexibility and customization over a traditional design approach.

Autonomic computing

Autonomic computing aims to simplify and automate the management of computing systems, both hardware and software, letting them self-manage without human intervention. To be self-managing, an autonomic system must have four characteristics:

- *Self-configuring.* The system must adapt automatically to its operating environment. Hardware and software platforms must possess self-representations of their abilities and self-configure with regard to their environment.

- *Self-healing.* The system must diagnose and solve service interruptions. It must recognize a failure and isolate it, thus shielding the rest of the system from its erroneous activity. It then must recover transparently from failure by fixing or replacing the section of the system responsible for the error.
- *Self-optimizing.* The system must constantly evaluate potential optimizations. Through self-monitoring and self-configuration, the system should self-optimize to efficiently maximize resources to best meet the needs of its environment and users.
- *Self-protecting.* The system must anticipate a potential attack, detect when an attack is underway, identify the type of attack (for example, denial of service or unauthorized access), and use appropriate countermeasures to defeat or at least nullify the attack.

All four characteristics (often collectively called self-* capabilities) involve the ability to handle functionality that has been traditionally a human system administrator's responsibility. The software domain has used adaptive and reflective techniques to empower systems to automatically self-alter (adapt) to meet their environmental and user needs. Such techniques already enhance several software services, including multimedia, security, transactions, and fault tolerance, and they point to a key emerging paradigm for the development of dynamic next-generation platforms.[1,4]

Initial implementations of self-managed systems have targeted specific domains and deployment.[5–8] So, their self-representations are specifically designed to tackle the requirements in the domain. These self-representations are thus tightly coupled to the system's implementation and the domain they describe.

If self-managed systems are to become part of standard industry practice, they'll need to cope with the everyday challenges of industrial environments. Such environments present diverse deployments and changing requirements. Systems frequently need to scale in both large multiserver distributed enterprise systems and small embedded devices and PDAs. Moreover, systems might be deployed across a diverse range of application domains, from payroll software to multimedia content delivery, with varying requirements. Successful operation in such environments will require flexible system implementations that are easily customizable to the target domain and associated requirements. The self-managed system and its self-representation for these environments must also be flexible and customizable to support these requirements.

Current design practices for self-representations focus on dividing system functionality into common fixed concerns to enhance usability and simplify implementation, with little consideration of flexibility.[5–8] Increasing flexibility necessitates addressing additional concerns specific to a self-representation implementation. The MVC design pattern is ideally suited to this task because it encapsulates the necessary crosscutting concerns of a self-representation in a straightforward, intuitive manner.

The role of a self-representation

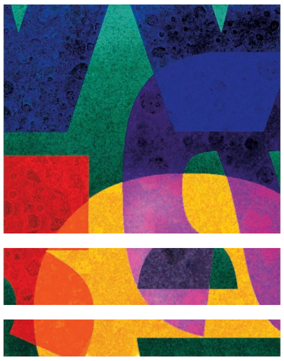
Reflection is a well-known self-management technique for providing principled mechanisms to inspect a system's structure and behavior. A reflective system maintains a representation of itself (self-representation), which is causally connected to the implementation of the underlying system and describes what that system does.[9] So, the underlying system behavior reflects changes made to the self-representation and vice versa. A self-representation plays an important role in the development of self-management capabilities. Inspecting and altering the self-representation lets self-management code or a system administrator examine system functionality and alter and reconfigure the underlying system's behavior. This can improve the system's performance in different contexts and operational environments.

To be effective, many current self-representations, including Open ORB,[6] dynamicTAO,[7] and K-Components,[8] have three operational roles:

- *State.* They can maintain information on the system's current state and the conditions experienced in its environment.
- *Analysis.* They can access state information and perform relevant examinations on it.
- *Realization.* They can alter the state information and update the system's functionality to express those changes.

Two design approaches have emerged for system self-representations. The first approach, which many early self-managed systems employed, incorporates the definition of the self-representation within the system functionality's implementation.[7] Although this approach is useful for leveraging existing code, mixing the self-representation with system functionality can lead to a complex implementation as self-management capabilities increase.

The second, more popular, approach separates



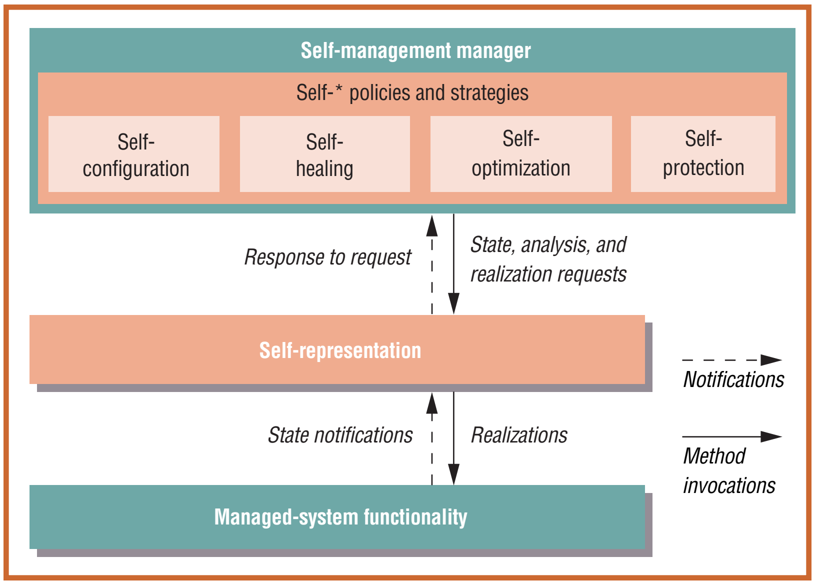


Figure 1. The architecture of a self-managed system. The self-management manager uses the self-representation to manipulate the managed system.

the system into two parts: system functionality and the self-representation. This approach is often called a *metaspace*,[10] *meta-architecture*, or *metalevel*. The base level provides system functionality, and the metalevel contains the self-representation along with policies and strategies for the system's behavior. The Metaobject Protocol (MOP) serves as the interface to the metalevel. To maintain clarity, we'll continue to use the terms *managed system functionality* (base level) and *self-representation* (metalevel) for the remainder of this article.

Figure 1 shows a typical design based on the dual-level approach. The self-management manager contains policies and strategies for the managed system functionality's self-configuring, self-healing, self-optimizing, and self-protecting behavior. On the basis of these policies and strategies, the manager can analyze and alter the system functionality by accessing the self-representation.

A well-designed self-representation tracks system functionality (security, distribution, fault tolerance, and so on) relevant to the system's self-management objectives. As the use of self-management capabilities increases, self-representations must track greater numbers of system concerns. To combat bloating and complexity, self-management designers separate system functionality concerns into multiple models, thus improving the self-representation's usability.[5] This technique has been successful at managing complexity and simplifying interaction. Initial implementations of this approach encapsulated system concerns within single objects. For example, Open ORB separated its self-representation into distinct objects for architecture, interface, interception, and resource system concerns.[6]

Inevitably, the size of a self-representation for a specific system concern will grow beyond a single object's practical size. Once this occurs, designers will need to consider the purpose of objects and interobject relationships and dependencies in the self-representation. In addition, current design practices provide minimal, if any, support to develop general-purpose self-representations that are customizable to a specific application domain or system implementation.

Gismo: A self-representation for MOM

Message-oriented middleware (MOM) is one of the foundations of distributed systems. Its uses in such systems range from providing small-scale communication infrastructure for embedded devices and PDAs to serving as the messaging backbone of massively scalable enterprise systems.[11] The Gismo (*G*eneric *S*elf-management for *M*essage-*O*riented Middleware) framework aims to provide general-purpose self-management capabilities for MOM systems.[12] The Gismo self-representation's implementation poses several challenges for contemporary design practices.

Challenges

Most MOM implementations share common behaviors and capabilities. However, they can also contain some form of proprietary functionality (message filtering, content-based routing, broker networks with varying deployment topologies, and so on). Additionally, MOM systems operate in a diverse range of application environments, from enterprise resource planning (ERP) systems to on-demand mobile multimedia platforms. For Gismo to be successful in each of these environments, its self-representation must track specific information on the resources and demands in the particular environment (data integration in ERP, quality of service for mobile video streaming, and so forth). A one-size-fits-all approach isn't appropriate; Gismo must be able to easily extend its self-representation in a controlled manner to include such information.

The design must not only meet these requirements but also do so in a way that doesn't negatively affect usability and that minimizes the effort required to port to new MOM implementations. We present a design for Gismo using current design practices, which illustrates their limitations.[5,6,8]

A contemporary Gismo design

Using current system concern separation practices for the initial design identified three distinct MOM system concerns, covering destinations, subscriptions, and interception activities. Destination concerns track the existence, basic configuration, and relationships between destinations (queue or topic) in the MOM. Subscription concerns monitor client activity by tracking message consumers' subscription details, including subscription constraints. Interception concerns involve enabling the dynamic insertion of interceptors. Interception is a vital technique for self-managed systems that offers a flexible mechanism for monitoring, altering, and extending the system's behavior at runtime. For instance, interception can inject functionality to execute every time the system adds a new subscriber or when an application sends a message to a client.

For brevity, we concentrate on the design of the destination self-representation. This self-representation has the three operational roles we mentioned before:

- *state*—data structure to track the type and basic configuration of destinations in the MOM;
- *analysis*—operations to examine destinations in the MOM (destination search, destination or subscription analysis, destination traffic analysis, and so on); and
- *realization*—an administrative interface for destinations, facilitating the creation, updating, and deletion of destinations and destination hierarchies in the MOM.

We implemented these roles using a collection of objects, as figure 2 shows.

This design decomposes the self-representation into three objects encapsulating the state, analysis, and realization roles. Many current designs encapsulate the state and analysis roles within a single object.

To fulfill their objectives, self-* policies and strategies inspect the state and analysis objects to monitor the system. For example, a self-optimizing strategy might use the destination analysis object to examine common subscription constraints for a topic destination. The strategy could then determine whether sufficient common constraints exist to justify the creation, using the destination realization object, of a new subtopic to optimize the MOM's performance.[13]

Design limitations

At first glance, the self-representation design might appear reasonable; a self-representation's three roles are encapsulated in distinct objects. However, closer inspection reveals several interdependencies that will increase the effort required to customize the self-representation. The GISMO self-management manager directly interacts with all three objects in the self-representation, and the managed MOM's functionality interacts directly with destination state and realization objects. Any changes to the destination state, analysis, or realization objects might require not only internal alterations of the self-representation but also external changes in both the GISMO self-management manager and the managed MOM functionality. This limits design flexibility.

Interaction with the MOM implementation crosscuts the self-representation. This coupling increases the effort required to change the self-representation and to port it to alternative MOM implementations. The self-representation design supports specific MOM implementations by implementing new state and realization objects. Although it's difficult to completely avoid the work required to port the self-representation, a clearer separation of concerns to improve encapsulation can minimize this effort considerably.

This example illustrates how the implementation of a self-representation design's operational roles can affect flexibility. It also shows that dependencies in the design don't promote customization or portability, and it illustrates the cost of change in such designs. As figure 3 shows, concern separation in a self-representation must consider operational concerns in addition to system concerns. Appro-

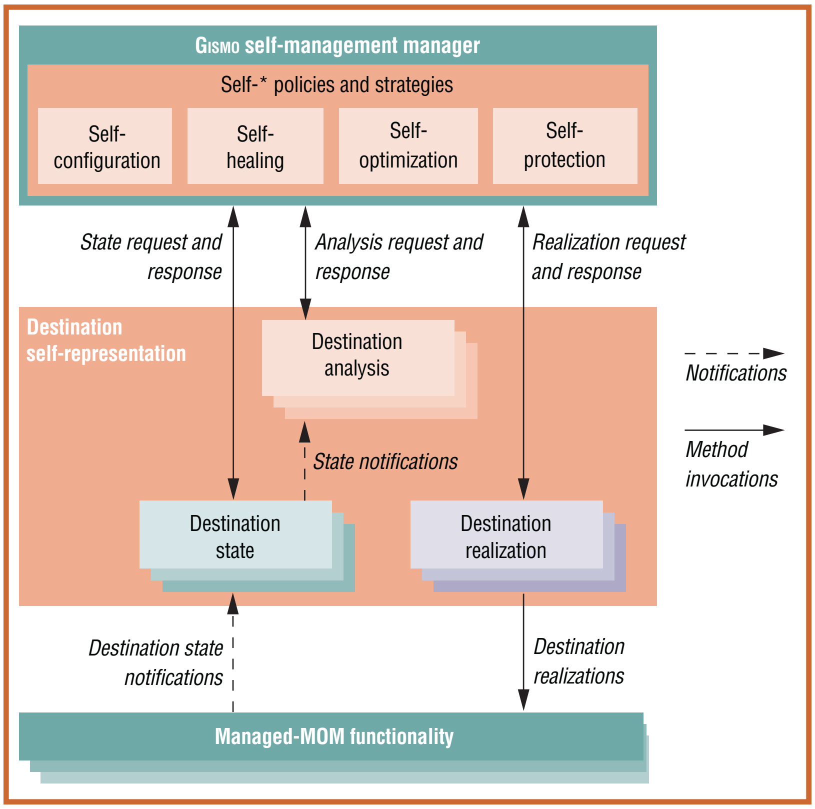

GISMO self-management manager
Self-* policies and strategies
Self-configuration
Self-healing
Self-optimization
Self-protection
State request and response
Analysis request and response
Realization request and response
Destination self-representation
Destination analysis
State notifications
Destination state
Destination realization
Notifications
Method invocations
Destination state notifications
Destination realizations
Managed-MOM functionality

Figure 2. Contemporary destination self-representation design. The self-management manager directly interacts with the state, analysis, and realization objects. The self-representation's state and realization objects interact with the managed system.

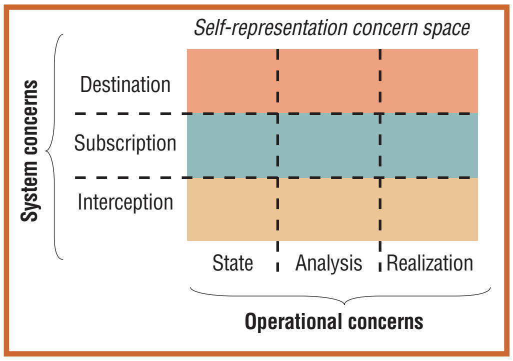


Figure 3. Operational-concern separation. The operational concerns (state, analysis, realization) of a self-representation crosscut system concerns. The design of a self-representation must consider their encapsulation.

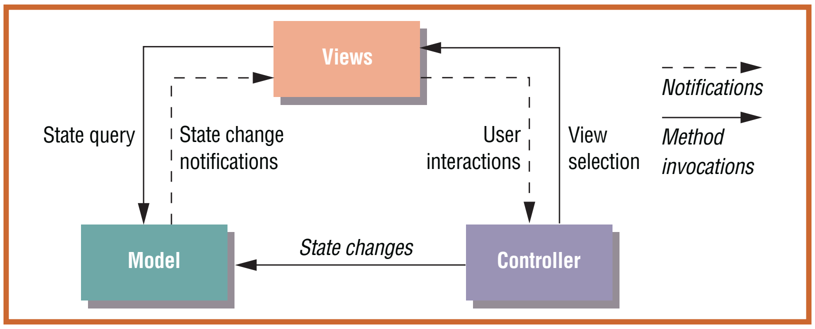


Figure 4. The Model-View-Controller (MVC) design pattern. The model contains the data, views present the data, and the controller processes events affecting the model or views.

priate separation of operational concerns can help reduce dependencies and promote flexibility.

The issues highlighted in the destination self-representation's design aren't unique in the software domain. In fact, one of the most successful design patterns—the Model-View-Controller (MVC)—solves these issues in regard to user interaction. Using a slight variation of this design pattern can improve concern separation in a self-representation in a straightforward, intuitive way.

The MVC

This design pattern has been very successful at concern separation for user interaction.[14] First described by Trygve Reenskaug in 1979 and implemented by Jim Althoff for Smalltalk-80,[15] the MVC separates user interaction from data processing, letting both change independently.

Pattern overview

The pattern achieves independence by decoupling data access, data-processing logic, and data presentation and user interaction tasks into three distinct object classifications. The model contains the data, views present the data, and the controller processes events affecting the model or views. Figure 4 illustrates the relationship between each MVC component.

The MVC offers a powerful mechanism for viewing and altering data. It lets a model have multiple views and controllers, which can be created and altered independently of the model. It facilitates the creation of highly flexible solutions and is prevalent in systems that must provide multiple views of the same data. Many of its benefits are equally applicable to the design of flexible self-representations.

The MVC in a self-representation

The MVC can improve operational-concern separation in a self-representation. The three main operational concerns of a self-representation map to the MVC pattern, letting it decompose operational concerns. First, the model contains representational state information in its most basic form, separating it from the realization and analysis operations.

Second, views create analyses of the representational state contained in the model. Views render the self-representation's state from a particular snapshot. There's no restriction on views' composition, and they can be created using a mixture of information from multiple model objects. Views can also perform computations on the model and augment it with additional external information sources, supporting a highly customized analysis of the self-representation. Views can also be temporal, tracking the changes to the model over a period of time.

Finally, in the design of a self-representation, controllers encapsulate all interaction with the underlying system functionality. The realization process requires direct interaction with the system functionality. So, controllers can be tightly coupled to a specific system implementation. The controller objects encapsulate this coupling and decouple the rest of the self-representation (model and views) from the system functionality's implementation. Multiple controllers can serve to realize the self-representation for alternative system implementations.

Figure 5 illustrates the MVC's role in a self-representation.

The controller encapsulates interaction between the managed system functionality and the self-management manager. Insulating the realization process produces a more controlled, looser coupling between the self-representation and the underlying system implementation. Changes to the model will no longer require reciprocal external changes in the managed system functionality and self-management manager. Instead, we use a slight variation of the traditional MVC, in which the controller updates the model only when the self-management manager requests a realization or when the controller receives a state notification from the managed system.

The qualities that make the MVC successful for user interaction also make it successful for designing self-representations. The clear concern separation introduced by the pattern improves the design by reducing dependencies and simplifying customization and portability.

MVC-based Gismo self-representation

The MVC-based design of the Gismo self-representation lets the latter meet its requirements in a straightforward manner. Figure 6 shows the design of an MVC-based destination self-representation.

This design decomposes the self-representation into three distinct object categories using the MVC pattern. The destination model tracks the managed MOM's destination state. Destination controllers encapsulate interaction with specific managed-MOM implementations and the self-management manager, which in turn uses the destination views to analyze the managed MOM's status.

We now examine this self-representation's merits on the basis of Gismo's requirements.

Customization

The first requirement is the ability to easily customize the self-representation to include proprietary MOM functionality, or information specific to the application environment, without affecting the self-representation's generality. Rather than a one-size-fits-all approach, the self-representation design lets designers tailor models, views, and controllers to specific system requirements encountered within their deployments. The MVC-based design allows new higher-level views of the representation, thus providing more user-friendly, application-specific adaptation. For example, the sequence of connect, delete, create, and disconnect required when dynamically replacing destinations can be encapsulated into a replace operation in a high-level application-specific controller. This ensures that the basic interaction of the self-representation will be customizable to improve usability.

Additionally, we can customize the representations to provide an appropriate set of operations, depending on the available resources in the deployment environment. Because Gismo can be deployed on resource-limited devices, we can tailor self-representations to reduce system resource usage—for example, to use only one representation at a time. We can easily customize a self-representation with domain-specific information to provide highly tailored analytical capabilities.

Portability

The second requirement is to minimize the effort required to add new MOM implementations. The

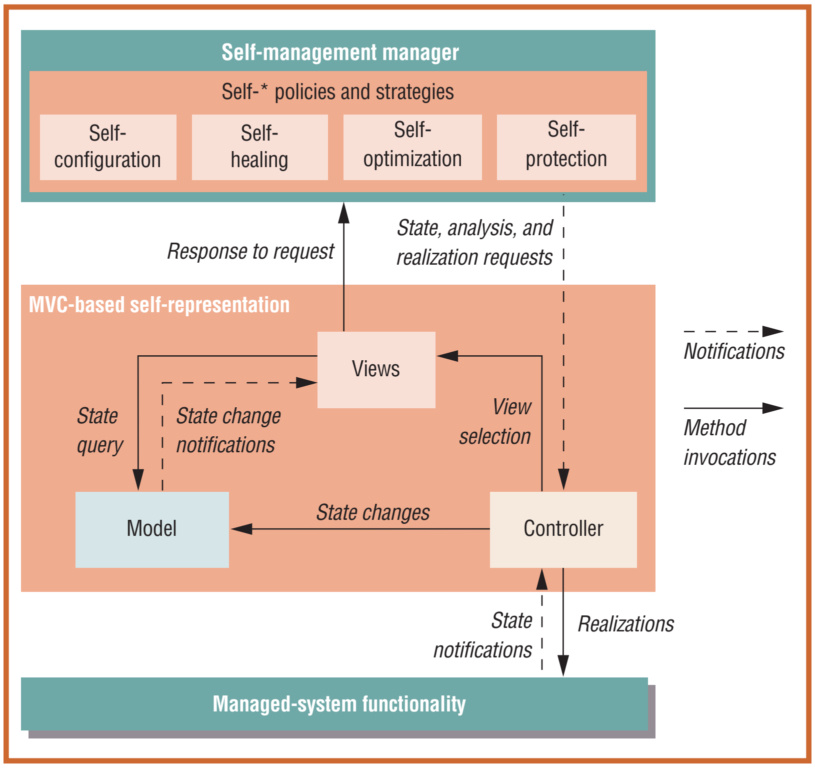


Figure 5. The MVC's role in a self-representation. The model contains the state; views provide the analysis; and the controller processes events affecting the model, views, or managed system.

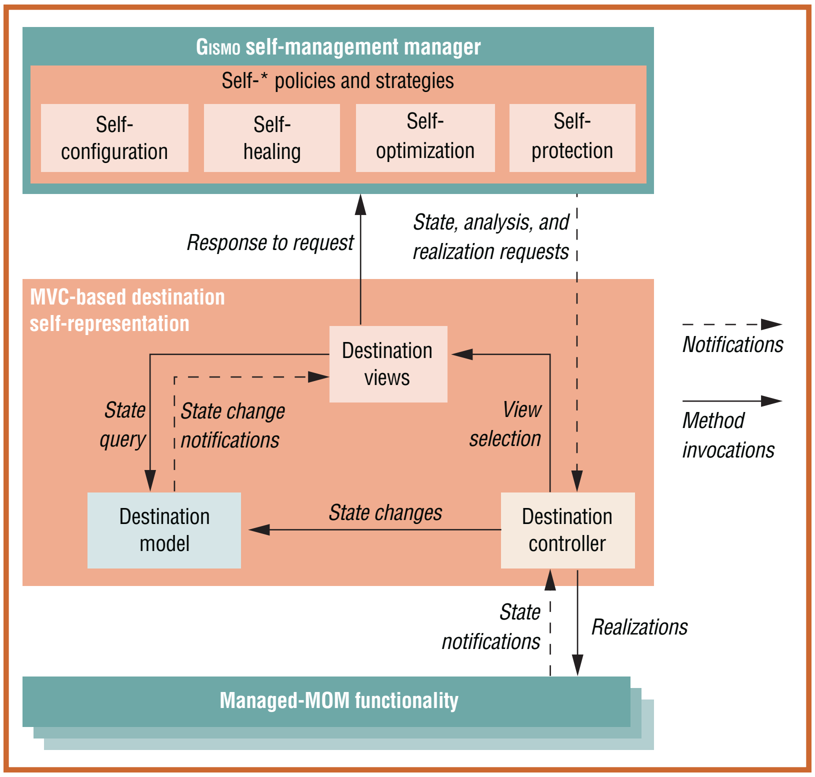


Figure 6. An MVC-based destination self-representation design. This design decomposes the self-representation into three distinct object categories using the MVC pattern.

About the Authors

Edward Curry is a researcher in the Digital Enterprise Research Institute at the National University of Ireland, Galway. His research interests include semantic search and self-* and nature-inspired middleware. He received his PhD in computer science from the National University of Ireland, Galway. He's a member of the IEEE and the ACM. Contact him at Digital Enterprise Research Inst., Nat'l Univ. of Ireland, Galway, University Rd., Galway, Ireland; edcurry@acm.org.

Paul Grace is a research associate in Lancaster University's Computing Department. His research interests include adaptive middleware, reflection, middleware for mobile and grid computing, and component programming. He received his PhD in computing from Lancaster University. Contact him at the Computing Dept., Lancaster Univ., Lancaster LA1 4WA, UK; p.grace@lancaster.ac.uk.

self-representation design encapsulates interaction with the managed MOM within the realization controllers. Porting GISMO to a new MOM requires only a new realization controller. New models and views are needed only when it's necessary to track state or functionality specific to the new MOM or environmentally specific information. The clear role separation for objects in the design also promotes reusability. In addition to the controller and views reusing the model, complex views can be created by reusing simple views.

We've also applied the MVC to the refactoring of the OpenCOM self-representation.[16] OpenCOM is a lightweight component model for developing adaptive systems software and has been implemented for Windows in C++, Linux in C, and also in Java. Each of these implementations tightly couples the self-representation to the underlying component's runtime kernel. We have used the MVC to allow customization of the self-representation and to simplify adding component types, such as Java Beans, COM (Component Object Model), and POJOs (Plain Old Java Objects). Reusing the existing models and views lets us support a new component type by simply implementing a new realization controller for that type.

Performing middleware adaptations using component-based self-representations can be difficult for nonexpert users because they must understand how the components implement the middleware. This situation can lead to verbose self-management code. So, we plan to investigate whether higher-level MVC-based self-representations for individual middleware types (such as a group communication representation) would simplify dynamic adaptation. Initial research in this area is producing promising results by making the Gridkit reflective-middleware framework customizable for different middleware solutions.[17] In addition, it's possible to extend self-representations in Gridkit to provide a richer set of operations, depending on available resources.

Acknowledgments

We thank the reviewers for their constructive comments, which have improved this article's quality.

References

1. K. Geihs, "Middleware Challenges Ahead," *Computer*, June 2001, pp. 24–31.
2. M. Weiser, "Some Computer Science Issues in Ubiquitous Computing," *Comm. ACM*, July 1993, pp. 74–84.
3. A. Ganek and T. Corbi, "The Dawning of the Autonomic Computing Era," *IBM Systems J.*, Jan. 2003, pp. 5–18.
4. R.E. Schantz and D.C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-Centric Applications," *Encyclopedia of Software Engineering*, John Wiley & Sons, 2001, pp. 801–813.
5. H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-model Reflection Framework," *Proc. Int'l Workshop New Models for Software Architecture (IMSA): Reflection and Metalevel Architecture*, ACM Press, 1992, pp. 36–47.
6. G.S. Blair et al., "The Design and Implementation of Open ORB 2," *IEEE Distributed Systems Online*, vol. 2, no. 6, 2001, http://csdl2.computer.org/comp/mags/ds/2001/06/o6001.pdf.
7. F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the DynamicTAO Reflective ORB," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing* (Middleware 00), LNCS 1795, Springer, 2000, pp. 121–143.
8. J. Dowling, "The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems," doctoral dissertation, Dept. of Computer Science, Trinity College Dublin, 2004.
9. P. Maes, "Concepts and Experiments in Computational Reflection," *ACM SIGPLAN Notices*, Dec. 1987, pp. 147–155.
10. G. Kiczales, J.D. Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1992.
11. E. Curry, "Message-Oriented Middleware," *Middleware for Communications*, Q.H. Mahmoud, ed., John Wiley & Sons, 2004, pp. 1–28.
12. E. Curry, "Increasing Flexibility within MOM Using Portable Rule-Bases," *IEEE Internet Computing*, Nov./Dec. 2006, pp. 26–32.
13. E. Curry, D. Chambers, and G. Lyons, "Reflective Channel Hierarchies," *Proc. 2nd Workshop Reflective and Adaptive Middleware, 4th ACM/IFIP/Usenix Int'l Middleware Conf.* (Middleware 03), LNCS 2672, Springer, 2003, pp. 105–109.
14. G.E. Krasner and S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *J. Object-Oriented Programming*, vol. 1, no. 3, 1998, pp. 26–49.
15. S. Burbeck, "Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)," 1992, http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.
16. M. Clarke et al., "An Efficient Component Model for the Construction of Adaptive Middleware," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms* (Middleware 01), LNCS 2218, Springer, 2001, pp. 160–178.
17. P. Grace et al., "Deep Middleware for the Divergent Grid," *Proc. ACM/IFIP/Usenix 6th Int'l Middleware Conf.* (Middleware 05), LNCS 3790, Springer, 2005, pp. 334–353.