Separate Application Logic from Architectural Concerns – Beyond Object Services and Frameworks

Zhenyu Wang
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
zwang@cs.cmu.edu

Abstract

Application logic refers to the encoding of domain business logic and architectural concerns refer to common non-business requirements on the system, for example, security, transaction, performance, scalability etc. It has been widely recognized that separating application logic from architectural concerns can greatly help reduce the development cost and make the system easier to evolve. Current state of practice relies on object services and frameworks. In this paper we argue why these technologies are not sufficient and propose a new approach based on architecture unification.

1. Separating Application Logic from Architectural Concerns

As part of a complete software system, application logic refers to the coding of business process for the problem domain, for example how money flows through a trading system. Usually besides application logic, business applications also require architecture level *functions* which do not show themselves in the business process, e.g. security and transaction, and *properties* such as reliability, availability, maintainability, responsiveness, manageability and scalability. A list of proposed architectural properties appear in [4].

Separating application logic from architectural concerns refers to the enabling technology to build common infrastructures to address architectural concerns. With this capability, developers can produce a complete system by only coding application logic and reuse those common infrastructures. This is desirable for several reasons,

- It will greatly reduce development cost. Engineers do not need to spend time on building infrastructures to support architecture functions and properties because these infrastructures can be reused from project to project.
- It makes systems easier to evolve. Requirement changes on business process and architectural properties can be treated separately. For example we do not need modify application logic code if a system does not scale well.

2. Architectural Functions and Properties

We observed that architectural functions and properties are related to following three system artifacts.

- Presence of service components. Usually service components will be introduced into the system for certain architecture function requirements. These components typically include authentication server, encryption/decryption library, dynamic load balancer and transaction monitor.

- Correct system configurations. Configurations refer to communication paths among system components, i.e. how components are glued together. Configurations are important for two reasons, first, to make right use of service components, the system must be configured correctly, second, some architecture properties are directly related to system configurations. For example, if all communications between clients and servers must go through a single security checker, system performance will suffer because of the bottleneck.

- Interaction Protocols. Usually system components interact with each other using proper protocols to provide certain functions. We are referring to application independent architectural protocols here. Typical examples include two-phase commit protocol used for distributed transaction coordination and various authentication protocols. Interaction protocols are important because, first, even with correct system configuration, we still need to make sure that components work in the right way, second, protocols themselves have influences on architectural properties like livelock free and performance. For example, protocols between clients and servers should require as few network trips as possible for performance reason.

3. Object Services and Frameworks

Lots of effort have been spent on building infrastructures to address architectural concerns. The current state of practice relies on following two technologies.

Object Services

Standard object services like CORBA object services have been proposed. Systems can be built as layered architectures where high level application logic components access low level object services transparently. This approach has been shown to be successful to a certain extent. For example, it has been shown in [3] that it is possible to build application-transparent security using DCE security service.

But object services only cover the first artifact we mentioned in section 2, developers still need to write code to correctly configure their systems and to access these services in proper protocols, for example they still need to make sure that client requests are evenly distributed among servers and still need to code the two-phase commit process. Usually these code are mingled with application logic code and are redeveloped again and again from project to project.

Frameworks

Frameworks are a promising technology for reifying proven software designs and implementations, usually these designs and implementations do capture architecture functions and properties. A framework is a reusable, semi-complete application that can be specialized to produce custom applications [7]. Typical framework examples are MacApp, MFC, Java RMI and CORBA.

Frameworks are able to capture three system artifacts mentioned in section 2 by acting as a backbone which connects service components and dictates proper system configurations. But the main problem with frameworks is that they are targeted at specific domains because usually part of the business logic is moved into frameworks themselves. As a consequence frameworks are very expensive to build and are not reusable across different domains [7].

So if we think of object services and frameworks as two ends of a spectrum**,** a better technology is something in the middle. We want to build reusable infrastructures to address architectural concerns without incorporating business logic.

4. The New Approach

We propose a new approach based on architecture unification. The basic idea is that we can specify *software architecture templates* which are very generic architectures with built-in architectural functions and proven architectural properties. Application logic can be developed independently and then unified with these templates to produce a complete system.

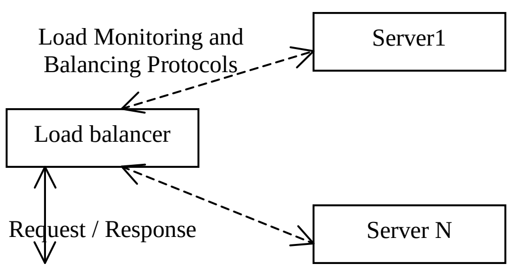
Software Architecture

Software Architecture [1] refers to the discipline of design and specification of system structure. Here structural issues include organization of a system as a composition of components, namely configuration, protocols for communication and architectural properties. Architecture Description Languages (ADLs) [8] have been proposed to formally describe a system architecture which includes components, configurations /communication paths among these components and interaction protocols. A*n architecture description easily covers the three system artifacts mentioned in section 2 and thus provides the right language for us to talk about architectural concerns*. Even more important, *architecture based analysis* can be used to prove properties of an architecture and it has been a very active research area within DARPA funded architecture research communities. There are several on-going research projects on analyzing security[2], performance[9] and dynamic properties[5] from an architecture description.

Architecture Templates

An architecture template consists of three elements, an architecture description, a set of preconditions and postconditions. The description specifies a generic (independent of individual applications) system architecture. Preconditions are assumptions made about properties of components and interactions protocols. Postconditions are properties of the architecture. Given a template, if preconditions hold for the architecture description then postconditions will hold for all systems which conform to that architecture. Preconditions and postconditions are usually deduced through architecture based analysis.

An architecture template example,



Precondition:

N < MaxServerConnectionNumber
Request Distrbution = model A
AverageServerResponse >= B

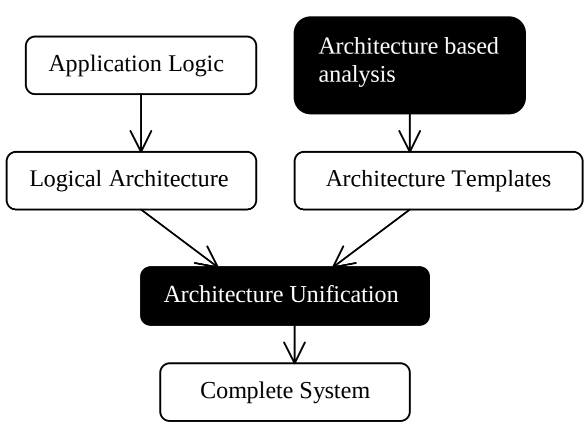
Postcondition:

Throughput >= C
ServerLoadVariation <= D

Basically this template specifies that if at most MaxServerConnectionNumber servers connect to a load balancer through a load balancing protocol and if request rate conforms to model A, then we can guarantee the throughput of the system to be at least C. In this example the architecture description is presented as a box-arrow diagram instead ADL description only for illustration purpose. Also the preconditions and postconditions may be too mathematical, in practice they will probably be written as simple rules instead of math equations.

The key point here is that these templates are very generic, they do not bind to business domains and yet they are powerful enough to dictates the presence of service components, proper configurations and interaction protocols which are directly related to architectural concerns. Because they are generic, they can be reused in a much larger context than frameworks.

Architecture Unification Process



Architecture unification is the process through which application logic gets integrated with architecture templates to produce a complete system.

Developers only need to define logical architectures which represent the application logic. A successful unification process will guarantee that all postconditions of architecture templates will hold, thus architectural properties of the final system will be guaranteed. Depending on the architecture functions and properties required, developers can choose what templates to use.

Intuitively the unification can be thought of as a mapping and incorporation operation. It consists of following five steps,

- Establish a mapping between logical architecture and a template. Components and communication paths in logical architectures will be mapped into their correspondents in the template.
- Validity Check. We need to make sure the mapping does not break configurations in the template and preconditions will hold for template components after the mapping. For example, if there is not a communication path between two components in the template, then the logical architecture should not have this communication path after the mapping either.
- Incorporating service components, e.g. performance monitors and load balancers which are invisible to logical architectures. They will be loaded into the final system.
- Incorporating configurations as indicated in architecture templates. For example a logical client server architecture may simply specifies that clients need to talk to a server module. And when we unify that architecture with the example template above and map the server module to all servers, the server module will be duplicated across those servers and connections with the load balancer will be set up.
- Architecture level interaction protocols will be integrated. For example, by mapping objects in logical architectures into transaction operations in a transactional architecture template, all operations in logical architectures will be wrapped around with transaction protocols automatically.

Due to the restriction on the size of this paper, I can not give more detail about the unification process. The key points here are, first, prechecked configuration rules captured by templates will be imposed on logical architectures, second, service components will be incorporated and proper access to these components will be set up automatically.

6. Conclusion

Architecture template unification is a promising technology to achieve separation of architectural concerns and application logic. On one hand, unlike object services, it captures configurations and interaction protocols, on the other hand, unlike frameworks, it does not commit too much to application logic. Of course, there are still many unresolved research issues around this technique, among them most important questions are, how much tool support can we provide for unification, what kind of preconditions and postconditions are useful in practice, how far can we go with architecture based analysis?

7. Acknowledgement

The position argued in this paper grew out of several discussions about component software architecture with David Garlan, whom I would like to thank.

Reference

[1] M.Shaw, D.Garlan, *Software Architecture*, Prentice Hall

[2] M.Moriconi, X.Qian, R.A.Riemenschneider and L.Gong, *Secure Software Architectures*, Proceedings of IEEE Symposium on Security and Privacy, May 1997

[3] J.Parodi and F.Burgher, *Integrating ObjectBroker and DCE Security*, Digitial Technical Journal, Vol 9, No 1, 1997

[4] Componentware Glossary, *http://www.objs.com/survey/ComponentwareGlossary.htm*

[5] R.Remi, R.Allen and D.Garlan, *Dynamic Architectures*, submitted for publication

[6] CORBA services, *http://www.omg.org/corba/csindx.htm*

[7] *Object-Oriented Application Frameworks*, ACM Communications, Number 10, Vol. 40

[8] N. Medvidovic and R. N. Taylor, *Architecture Description Languages*, Software Engineering ESEC/FSE 97, Vol. 1301 Lecture Notes in Computer Science, Zurich, Switzerland, Sep. 1997. Springer

[9] B. Spitznagel and D. Garlan, *Architecture Based Performance Modeling*, submitted for publication