

◆ Using Design Patterns to Build a Framework for Multimedia Networking

Just A. van den Broecke and James O. Coplien

A pattern-based approach to designing software offers enormous advantages for keeping pace with the rapidly changing field of multimedia networking. MediaBuilder, a framework for networked multimedia applications, demonstrates the power of this approach. This paper describes the main patterns that contribute to MediaBuilder's object-oriented architecture, revealing the framework's structure and the forces that shaped it. It then shows the effectiveness of the framework by presenting a conferencing application built with MediaBuilder that adapts readily to different multimedia and network standards.

Introduction

Advances in broadband networking and multimedia computing are expected to open a wealth of new applications, such as teleshopping, video on demand, and teleconferencing. Although many exciting new services will enter our workplaces and living rooms, providers and manufacturers face unknown challenges in the areas of:

- Application/end-user services,
- Network and multimedia standards, and
- Hardware platforms and operating systems/environments.

The future, however, is unpredictable, so definitive requirements are not available. How can we design our products to allow each area to evolve gracefully and independently?

Distributed system services are applying standard programming interfaces and protocols to help facilitate heterogeneity in operating systems, networks, and hardware devices. Standard programming interfaces make it easier to develop applications for a variety of platforms, and standard protocols enable programs to operate together. These distributed system services are called *middleware*,¹ because they sit “in the middle,” layering above the operating system and networking software and below specific applications. Middleware can offer a set of general-purpose services, such as

remote procedure call (RPC) and messaging, and may be specific to a particular domain, such as distributed databases or network management. Middleware for applications that combine networking and multimedia (audio, video, and data) is a topic of active research^{2,3} and standardization.^{4,5}

MediaBuilder is a middleware framework that targets the domain of networked multimedia applications. It supports areas such as session management, application protocols, and multimedia devices. Applications can be created by extending the framework and/or by reusing components developed with the framework. MediaBuilder was developed by the PLATINUM project (see **Panel 2**). Within this project, MediaBuilder was used to build a teleconferencing application that runs over an asynchronous transfer mode (ATM) network infrastructure.

The MediaBuilder software architecture is shaped by a collection of cooperating software patterns. *Patterns*—reusable microarchitectures that contribute to an overall architecture⁶—are an emerging discipline that helps designers understand and apply proven solutions to recurring design problems. The body of literature for the patterns of object-oriented programming,⁷ telecommunications architecture,^{8,9} and many other software areas is growing.

Panel 1. Abbreviations, Acronyms, and Terms

API—application programming interface. Programming-level functions providing the services of a software component.

ATM—asynchronous transfer mode. High-speed packet switching technique with fixed-size packets.

B-ISDN—broadband integrated services digital network

CM—conference management

CORBA—Common Object Request Broker Architecture. Standard for distributed object technology defined by the Object Management Group.

DLL—dynamic link library

FSM—finite state machine

GlobeView[®]-2000—ATM switch marketed by Lucent Technologies

GUI—graphical user interface

ISDN—integrated services digital network

ITU—International Telecommunications Union

middleware—Distributed system services provided through standard application programming interfaces

MM—multimedia. The meaningful integration of multiple media types with at least one time-dependent media type.

N-ISDN—narrowband integrated services digital network

OMG—Object Management Group. Industrial consortium that defines standards for object technology.

OS—operations systems

POSA—Reference to the book *Pattern-Oriented Software Architecture*.¹⁰

QoS—quality of service. Quality of a network path expressed by parameters such as cost, delay, probability of loss, probability of undetected errors, and throughput.

RPC—remote procedure call. Calling a function on a remote system as if it were a local function call.

TCP/IP—transmission control protocol/Internet protocol

TINA-C—Telecommunications Information Networking Architecture Consortium. International consortium that defines an open architecture (TINA) for telecommunications systems for the broadband, multimedia, and information era.

UML—Unified Modeling Language

Winsock2—An application programming interface for transparent network programming using Microsoft Windows.*

In this paper we focus on the main patterns that contribute to the MediaBuilder architecture. We show how patterns can be used to address design issues within the domain of networked multimedia applications and how they can be woven together to shape an architecture. The description of the MediaBuilder architecture, therefore, is illustrative rather than exhaustive.

The remaining sections are divided as follows: “Patterns” defines what patterns are, presents an example, and describes how to use patterns to build a framework. “MediaBuilder Architecture” shows how existing and new design patterns are woven together to address the issues affecting multimedia networking. “Application Example” presents a conferencing application built with MediaBuilder. “Conclusions” summarizes this paper’s findings, and “Future Directions” projects how MediaBuilder will evolve.

Patterns

In this section we define patterns, their form, and the relationship between patterns and frameworks. We show how a pattern can generate a microarchitecture from a general-purpose design pattern that meets an important application need. The notation used throughout this paper is also presented.

What are Patterns?

Patterns support a problem-solving discipline with rapidly growing acceptance in the software architecture and design community. A *pattern* is a structured document that describes a solution to a problem in a context. It captures the key design constructs, practices, and mechanisms of core competencies such as object-oriented development or fault-tolerant system design. Not necessarily object oriented, patterns supplement general-purpose

Panel 2. The PLATINUM Project

The PLATform providing Integrated services to New Users of Multimedia (PLATINUM) project had two main goals: (1) to build a high-speed network based on switched ATM, and (2) to develop distributed multimedia applications that use this network.

ATM technology is already deployed in the backbone of existing networks. The goal of the PLATINUM project was to make native ATM available to desktop computers via signaling. This would enable applications to request and use connections of bandwidths as high as 155 Mb/s. End users could experience multimedia services such as teleconferencing with high-quality audio and video.

Figure 1 shows a simplified picture of the PLATINUM platform. The cloud in the middle currently contains a single GlobeView®-2000 ATM switch developed and manufactured by Lucent Technologies. The two terminals shown in the figure are standard PCs running Microsoft Windows NT.* The three functional layers shown are the network layer, the MediaBuilder middleware layer, and applications.

The network layer allows users to establish ATM connections by providing ATM signaling protocols (terminals and the switch) and call processing (the switch). The PLATINUM project makes it possible to

negotiate connection characteristics such as bandwidth and connection topology (for example, multi-point for conferencing) through the switch.

The MediaBuilder middleware layer makes applications oblivious to underlying networks and provides components for application creation. It offers applications a generic session management service that adapts to a variety of networks, such as ATM (used in the project) or the Internet. Reusable components are provided for a variety of user protocols and for the local mapping of media streams to multimedia devices such as sound cards and cameras. The applications also allow end users to share media such as audio, video, whiteboard, and documents.

The project was conducted between January 1995 and July 1996. Contributors from industry, research institutes, and universities included Lucent Technologies (The Netherlands and the U. S.), AT&T Global Information Solutions (GIS) Germany (now NCR), Deutsche Telekom Germany, Telematics Research Center (TRC, The Netherlands), and the Center of Telematics and Information Technology (CTIT, The Netherlands). PLATINUM was built on earlier work done in the European RACE program¹⁵ and AT&T FastTrack. See also Klapwijk et al.¹⁹ and Ouibrahim and van den Broecke.²⁰

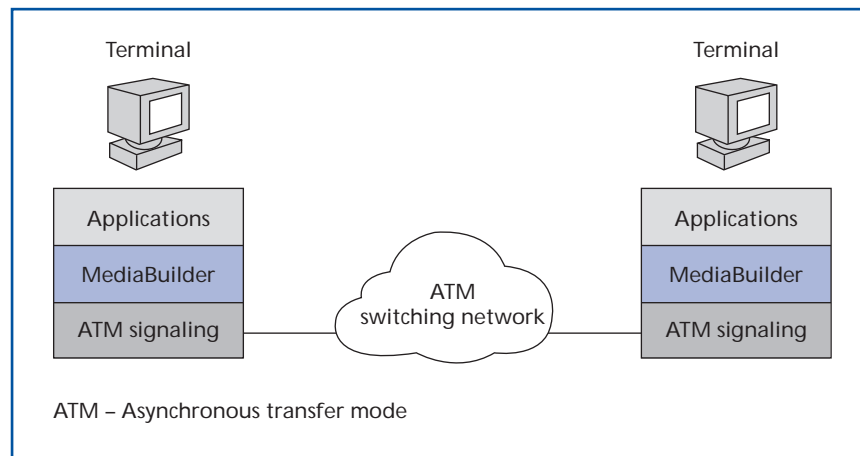


Figure 1.
Network configuration for the PLATINUM project.

design by capturing expert solutions in a form that helps developers solve difficult, recurring problems. Patterns are being widely used for teaching, for docu-

mentation, and as design aids.

A pattern describes the core structure of a solution at a level high enough to generalize to many specific

situations. Just as a dress pattern can be used to cut out many dresses of slightly different style, size, and trim, so a software pattern is a general solution that can be tailored to fit. For example, the *Observer* pattern, presented in the next section, describes a *dependency registration* structure, in which one piece of software is affected by changes that take place in another piece of software. Many aspects of the solution in this widely recurring software problem are subtle and difficult for programmers to solve. The *Observer* pattern describes the core elements of a working solution, but still leaves many implementation details to the discretion of the developer. Patterns tell the developer what to do without specifying how to do it; they are abstract, yet not vague.

A pattern has several sections beyond the solution. It describes the problem being addressed and the context in which it arises. It also describes the design tradeoffs—called *forces*—that it strives to balance. A pattern provides a *rationale*, which explains how the forces are balanced, and a *resulting context*, which describes new problems that arise because of the pattern or special situations that call for attention to other patterns. Together, these sections define a *pattern form*. Several popular pattern forms exist, all of which have these basic elements, but each of which adds other elements, such as applicability and intent, to emphasize specific design concerns. This form helps developers understand the problem, the solution, and the relationship between them, which enables developers to tailor an appropriate solution. A pattern is much more than a simple pairing of problem to solution.

The term *design pattern* usually refers to a collection of 23 patterns described in a book by Gamma, Helm, Johnson, and Vlissides,⁷ also called the “Gang of Four” patterns, referring to the four authors. These patterns follow the GOF form, which is widely emulated in the industry. They describe design solutions to problems that arise in object-oriented design. The patterns are largely independent of programming language, tools, and design methods, yet are specific enough for a novice programmer to implement. Familiarity with GOF patterns is, in many respects, a measure of competence in object-oriented programming.

The patterns described in this paper are related to object-oriented design. Some are GOF patterns that we have applied to the domain of multimedia networking. A recent book by Buschmann et al., of Siemens,¹⁰ provided a valuable source for several other patterns that we have used. It has been dubbed the “POSA” book, an acronym of its title, *Pattern-Oriented Software Architecture*. The pattern form we use is described in “Pattern Form and Notation,” later in this section.

The *Observer* Pattern

The *Observer* pattern, from the GOF collection, serves to “define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”⁷ The pattern is popularly used in human/machine interface designs to keep the user interface current with values of variables in a program. But that is just one specific use of *Observer*; the pattern can be generalized to fit many dependency registration problems.

Most good patterns have strong visual analogies. **Figure 2** shows the structure of a typical *Observer* implementation (see “Pattern Form and Notation,” later in this section, for the notation used). The intent of the pattern is to keep the state of `ConcreteObserver` consistent with that of `ConcreteSubject`. The structure uses general-purpose abstract base classes `Observer` and `Subject` to decouple the dependency registration logic of the two classes. When `ConcreteSubject` goes to a new state, it notifies its observers (`Notify()`) so they can make their states consistent with the change. The notification takes place through the `Notify()` interface of the abstract `Subject` base class. The `ConcreteObserver` may then use `GetState()` to query the `ConcreteSubject` for state change details.

If we captured *Observer* in a higher-level design abstraction—for example, as a set of C++ abstract base classes—we would obscure important design components and design relationships. The purpose of a pattern is not to hide design secrets, but to clarify the relationships between the parts. Furthermore, a given software artifact such as a function or class may participate in several patterns at once; the partitioning is not strictly hierarchical. A pattern is sufficiently general that no single implementation may generalize to meet

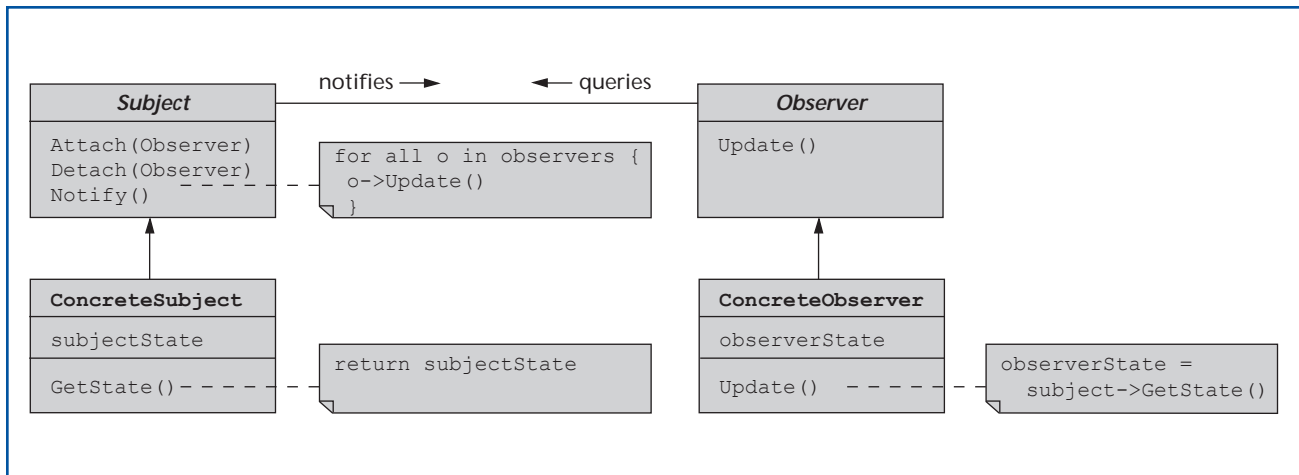


Figure 2.
The Observer pattern.

developer needs, so it is counterproductive to make it fit a single context.

Observer might be reused a million times without being implemented exactly the same way twice. If we use one invocation to propagate the notification event, followed by a reciprocal request for detailed state change information, the two actions can be combined by providing state information as an argument to the `Notify()` operation of the *Observer* class. This is not a new pattern, but rather a variant of the same pattern.

Building Frameworks with Patterns

We can think of some patterns as abstract descriptions of small software frameworks. A *framework* is a partial program for a problem domain. A developer can extend a framework at its external interfaces to build a complete, running application. Most object-oriented programming languages are rich in constructs to extend existing abstractions. For example, *inheritance* allows the designer to incrementally define a new type (for example, class) in terms of an existing one. Templates (sometimes called parameterized types) allow even more general-purpose parameterization of algorithms, user-defined types, and data structures.

Patterns are more abstract; they describe the major components and the relationships among them without the detail of coding mechanisms. As we described earlier for the *Observer* pattern, one pattern may generate many viable implementations. A good pattern factors common changes from the recurring, stable

structure. A developer should be able to conveniently capture specific application needs in new code, and then to just as conveniently hook this code into the framework using mechanisms supported by the programming language. One thing that keeps patterns abstract is their ability to remain flexible in application-specific variations.

A single pattern can generate a microarchitecture that can serve as a reusable framework. For example, a developer might use the *Observer* pattern as the architectural foundation for network conferencing. A state change in one terminal usually implies state changes or actions at other terminals. If one terminal goes mute (temporarily turns off its audio channel), other terminals may want to display an icon identifying the terminal that is muted. If one terminal goes on hook (hangs up), other terminals must be told that they should no longer try to communicate with that terminal. We can think of this use of *Observer* as a conference management framework that can easily accommodate many classes of change, such as the addition of new terminals or terminal types, the way call states are computed or represented, and many others. This architecture separates the addition of new features (such as private subconferences) from the underlying framework common to all features.

Most useful frameworks, such as the ET++ framework,¹¹ are rich in patterns, combining many microarchitectures into a larger structure. The resulting framework can be documented by new

Panel 3. Unified Modeling Language

Jointly, Grady Booch,²¹ James Rumbaugh et al.,²² and Ivar Jacobson et al.²³ have recently started to define the Unified Modeling Language (UML),¹² which is expected to become a main industry standard. We use a subset of the UML notation, whose notation elements are summarized below.

Figure 3 shows a class diagram. Classes are depicted as rectangles, each of which includes three compartments—the class name, attributes, and operations, respectively (see *SessionComponent*). Class names are shown in bold, abstract class names in bold italics. The compartments for attributes and operations

are optional. Inheritance is indicated by a solid arrow from the subclass to the superclass (for example, *Party* inherits from *SessionComponent*). General associations between classes are drawn as solid lines, optionally labeled with the direction and nature of the association (*SessionComponent* notifies *SessionObservers*). An aggregation, or “whole-part,” relationship is shown as an association with a diamond at the class that is the container (*Session* contains *Party*). Comments are depicted by a rectangle with a flipped corner and a dashed line connecting it to the commented element.

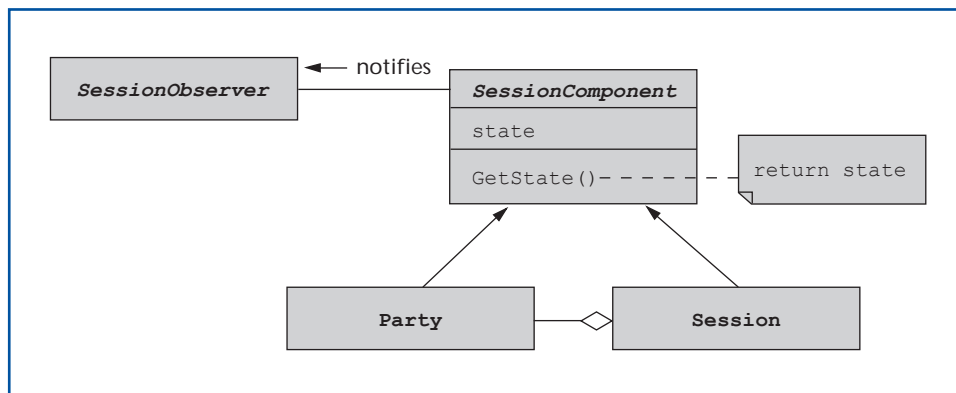


Figure 3.
An example of a class diagram in UML notation.

patterns that describe how to extend it for specific desired behaviors.

Pattern Form and Notation

Class diagrams drawn in this document use the Unified Modeling Language (UML) notation¹² (see **Panel 3**). We also introduce an extension of the UML notation to show patterns together with classes (see **Panel 4**). Pattern names are set in italics, and class names are set in the Courier type font.

The pattern form we use in this paper contains the following sections:

- *Name*—The name of the pattern.
- *Context*—The situations in which the pattern may apply.
- *Problem*—The problem that the pattern addresses.
- *Forces*—Constraints related to the problem and/or tradeoffs shaping the solution.

- *Solution*—The principles of the solution.
- *Resulting context*—The benefits and liabilities of applying the pattern.
- *Examples*—Where and how the pattern has been applied within MediaBuilder.
- *See Also*—References to related patterns.

MediaBuilder Architecture

This section shows how existing and new design patterns address issues within the domain of multimedia networking and how these patterns are woven together to shape the MediaBuilder architecture.

Purpose and Services

Consider a desktop telephony application. Typically, this type of application provides connection control and transport of audio data streams. These streams are mapped locally to an audio device, such as a sound card. A more advanced example is a tele-

Panel 4. Pattern Modeling Notation

Booch describes a potential extension to the UML to include patterns.²⁴ A pattern can be regarded as a template or a recipe to generate an instance of the classes that a designer uses in his or her design. These classes will conform to the roles of the pattern's classes. Patterns are drawn as ellipses. A designer's class that plays a role in a pattern is depicted by a dashed arrow pointing from the class to the pattern, labeled with the role being played.

We have also used this notation to show relationships between patterns. First, a role within a pattern may be played by another pattern, shown by the same notation, that is, a labeled dashed arrow from the pattern that conforms to a role. Second, a pattern can be a variant of a more abstract or better known pattern in a different domain, shown by a

dashed arrow from the variant pattern to the base pattern. No label is used because the entire variant pattern conforms to the base pattern.

Figure 4, excerpted from the MediaBuilder architecture, illustrates the notation. The pattern *Session Control & Observation* conforms to the *Model View Controller* pattern. The *Session Control & Observation* pattern has three participants: *SessionControl*, *SessionModel*, and *SessionObserver*. The class *SessionControlCommand* plays the *SessionControl* role and the *Command* role within the *Command* pattern. The pattern *Parties & Media as First Class Citizens* plays the *SessionModel* role. The class *MediumBuilder* plays a *SessionObserver* role. As used here, conformance is not the same as inheritance, but rather compliance with the behavior of the pattern or one of its participant's roles.

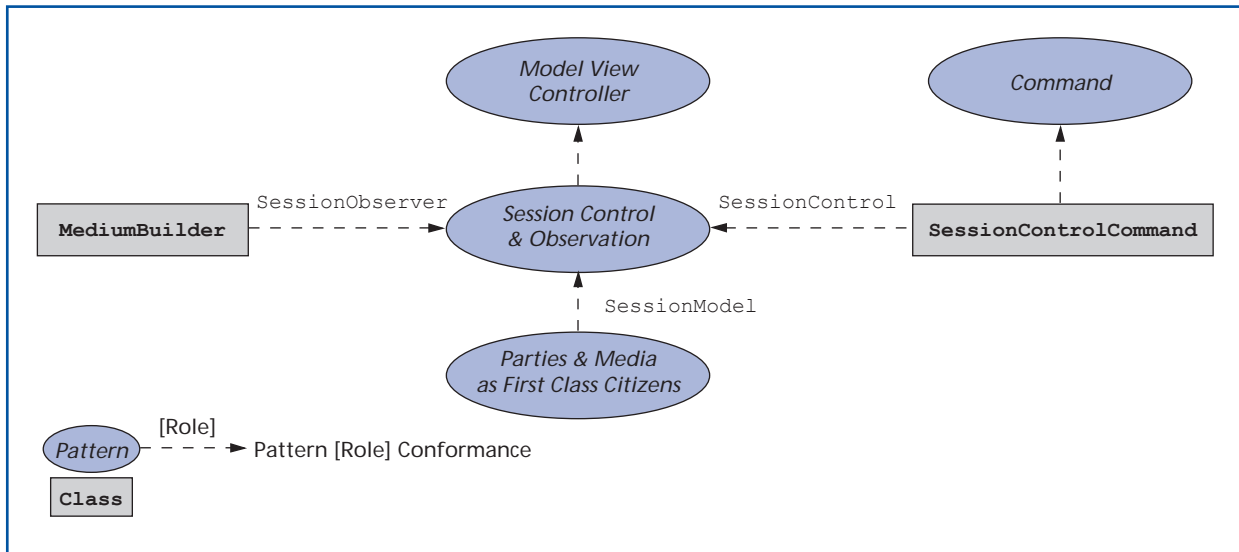


Figure 4.
UML notation extended for design patterns.

conferencing application. In this case, multiple applications cooperate over the network, sharing media such as video, audio, and whiteboard. In these media, the connection procedures and connection topologies (for example, multicast) are more complex. Multiple media streams have to be locally mixed (audio) or composed (video) and mapped to their corresponding multimedia devices. The sophistication of the underlying network protocols and ser-

vices, such as bridging, determines the amount of work an application must perform.

MediaBuilder allows software designers to create networked multimedia applications rapidly. It offers both a framework and a set of reusable components developed with that framework, as shown in **Figure 5**. Applications can be created by extending the framework and/or by selecting from components.

Functionally, MediaBuilder is located between

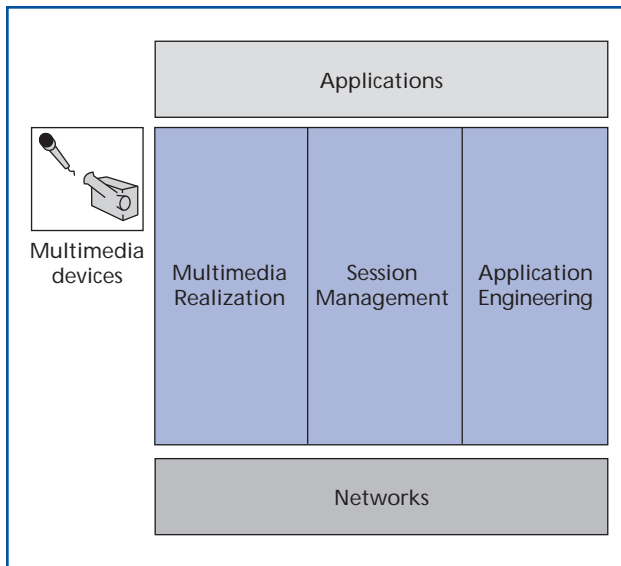


Figure 5.
A functional overview of the MediaBuilder architecture.

end-user applications, networks, and multimedia devices. Its applications can be used on different types of networks and their associated protocols. For example, the current MediaBuilder implementation runs on both ATM signaling networks and the Internet.

The services that MediaBuilder provides fall into three functional areas. Session management provides services that enable applications to control communication sessions. Multimedia realization deals with user protocols/multimedia streams and their mapping to local multimedia devices. Application engineering provides services to configure and instantiate framework components. The sections that follow discuss each functional area in more detail.

Session management. Session management allows applications to establish a shared context. Applications modify and negotiate this context through the session management application programming interface (API). This API provides a high-level service that is independent of the underlying network. API services are used to:

- Set up/release a session,
- Add/delete parties and/or media,
- Join/leave a session,
- Reinstate a session,
- Modify characteristics of media and/or parties,
- Support roles and permissions of parties with respect to session management,

- Contact directory services, and
- Respond to incoming requests.

The two main functions of session management are:

- Mapping between user quality of service (QoS), such as sharing a video medium, and a network QoS, such as bandwidth and multicast; and
- Negotiating the session context with peer session management entities.

Multimedia realization. Multimedia realization carries out the context negotiated by session management. This involves establishing transport connections for the end-to-end exchange of multimedia data, application protocols, and local mapping on multimedia devices. Initially, session management establishes an agreement between two users to share, for example, audio and video facilities, and then multimedia realization makes the connections that allow them to see and hear each other.

A set of base classes for application protocols, object communication, and finite state machines (FSMs) help develop new protocols and handle real-time sensitive streams. Extensible adapter classes provide a variety of standard transport services, such as native ATM or transmission control protocol/Internet protocol (TCP/IP). Services for buffering, splitting/combining, and multithreading support real-time streams, such as audio and video.

Multimedia devices—video cameras, speakers, microphones, audio files, and windows to display a shared whiteboard, and others—are producers and/or consumers of multimedia information. Base class abstractions are used to develop multimedia devices that support specific hardware or operating systems (OS)/multimedia interfaces.

Application engineering. Components developed in each area described above can be reused. Reusable components may be individual classes or more extensive “building blocks” that, when selected and combined, can create (engineer) applications. Application engineering is supported by:

- A repository of available components,
- Configuration services to select and combine components, and
- Run-time mechanisms that allow components

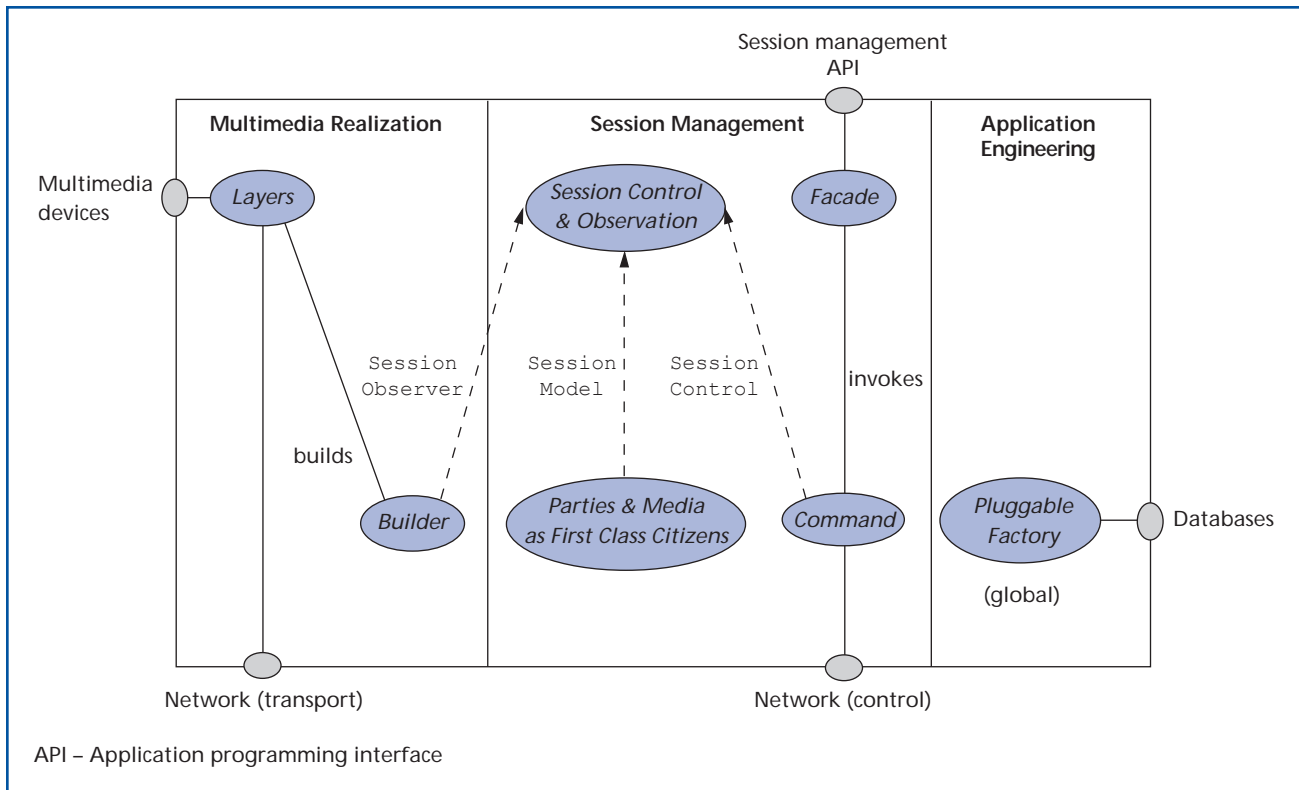


Figure 6. Architecture of MediaBuilder patterns.

to be dynamically linked, created, and attached to the framework.

Components available for reuse include audio, video, shared whiteboard, reliable multicast, shared object protocol, ATM signaling control, and conference management.

Architectural Overview

This section summarizes the seven main patterns of the MediaBuilder framework, which are described in detail in subsequent sections.

Figure 6 shows the main design patterns, their relationships, and their external interfaces (small gray ellipses). Each pattern belongs to one of the functional areas introduced above—session management, multimedia realization, or application engineering. Although many more patterns contribute to the architecture, these seven patterns form the backbone of the MediaBuilder framework and determine its basic behavior.

The *Facade* pattern⁷ is applied within MediaBuilder to give applications a common, high-level interface, the

session management API, while shielding clients from the details of how a session is controlled.

The *Command* pattern⁷ encapsulates a request as a class. It is applied within MediaBuilder to execute network-specific control procedures through which a session context is negotiated.

The *Session Control & Observation* pattern controls and maintains the state of a session and notifies those interested in that state. The pattern's three participants are realized using additional patterns, as shown in **Figure 3**. The session state (*SessionModel*) is modeled using the pattern *Parties & Media as First Class Citizens*. *SessionControl*, which controls the session state, is realized using the *Command* pattern.⁷ *SessionObservers* have an interest in the session state.

The *Builder* pattern⁷ is a specialized *SessionObserver* applied to build a realization of a session context (the *SessionModel*) that is local to the application. This involves creating and linking transport connections, application protocols, and local

multimedia devices. These elements are structured according to the *Layers* pattern.

The *Pluggable Factory* pattern provides a global service for configuring and instantiating framework components. A database holds a repository of components that can be reused in different applications.

Session Management Patterns

The four patterns that contribute to session management are *Session Control & Observation*, *Parties & Media as First Class Citizens*, *Facade*, and *Command*.

Session control & observation. MediaBuilder separates session control (session management) from session usage (multimedia realization). This concept is similar to the separation between control and user planes in the integrated services digital network (ISDN). Using the *Session Control & Observation* pattern, MediaBuilder links session management and multimedia realization.

Name

Session Control & Observation.

Context

Advanced signaling protocols, such as in narrow-band ISDN (N-ISDN) and broadband ISDN (B-ISDN), separate control protocols (in the C plane) from user protocols (in the U plane). C plane protocols negotiate characteristics of U plane protocols. Clients such as terminal applications and switch call processing will have to integrate actions in these two planes. In addition, clients often need to maintain complex state information related to entities such as calls, parties, and connections. This information may be used for billing, connection modification, and status display.

Problem

How should responsibilities between classes that implement control and usage functions be decoupled?

Forces

- Usage functions should be triggered from control functions.
- A direct coupling between control and use inhibits these functions from evolving independently.

Solution

Divide these responsibilities among three classes: `SessionModel`, `SessionObserver`, and `SessionControl`. `SessionModel` maintains the

state and notifies `SessionObservers` of state changes. `SessionControl` executes network-specific control procedures using `NetworkControl` on behalf of a `Client`. `SessionControl` reflects the (intermediate) results in the `SessionModel`. `SessionObserver` acts on state changes from the `SessionModel`. Specialized `SessionObservers` can handle the (instantiation of) protocols in the U plane or other usage functions such as billing or status display. **Figure 7a** shows an example of the pattern's class structure.

This solution is a specialization of the *Model-View-Controller* pattern.¹³

Examples

The message trace diagram (**Figure 7b**) shows a sample of how the pattern is used in an N-ISDN application. The `SessionModel` consists of a single `Call` object. `SessionControl` is represented by the object `OutgoingSetup` and interacts with `Call` and a `Q931API` object (a `NetworkControl`). Two `SessionObservers` are shown: a `SpeechApplet`, which is responsible for handling audio functions (transport and presentation), and a `StatusDisplay`, which graphically monitors the call state.

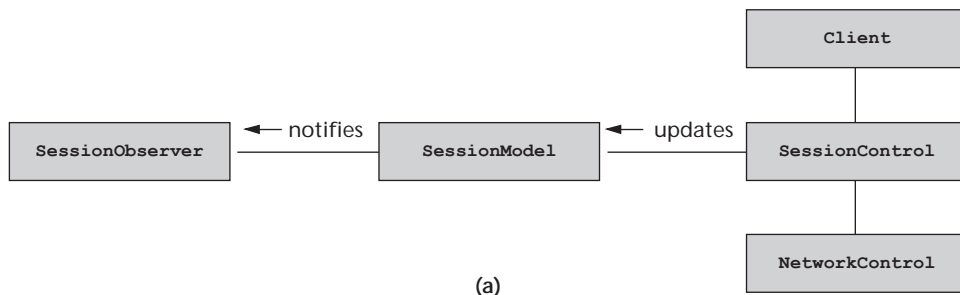
Resulting Context

The pattern integrates control and usage aspects for sessions, while allowing these functions to evolve separately. For example, new `SessionObserver` types may be developed and attached to the `SessionModel` without a need to change `SessionModel` or `SessionControl`. Similarly, `SessionControl` may be updated by additional control procedures or architectures.

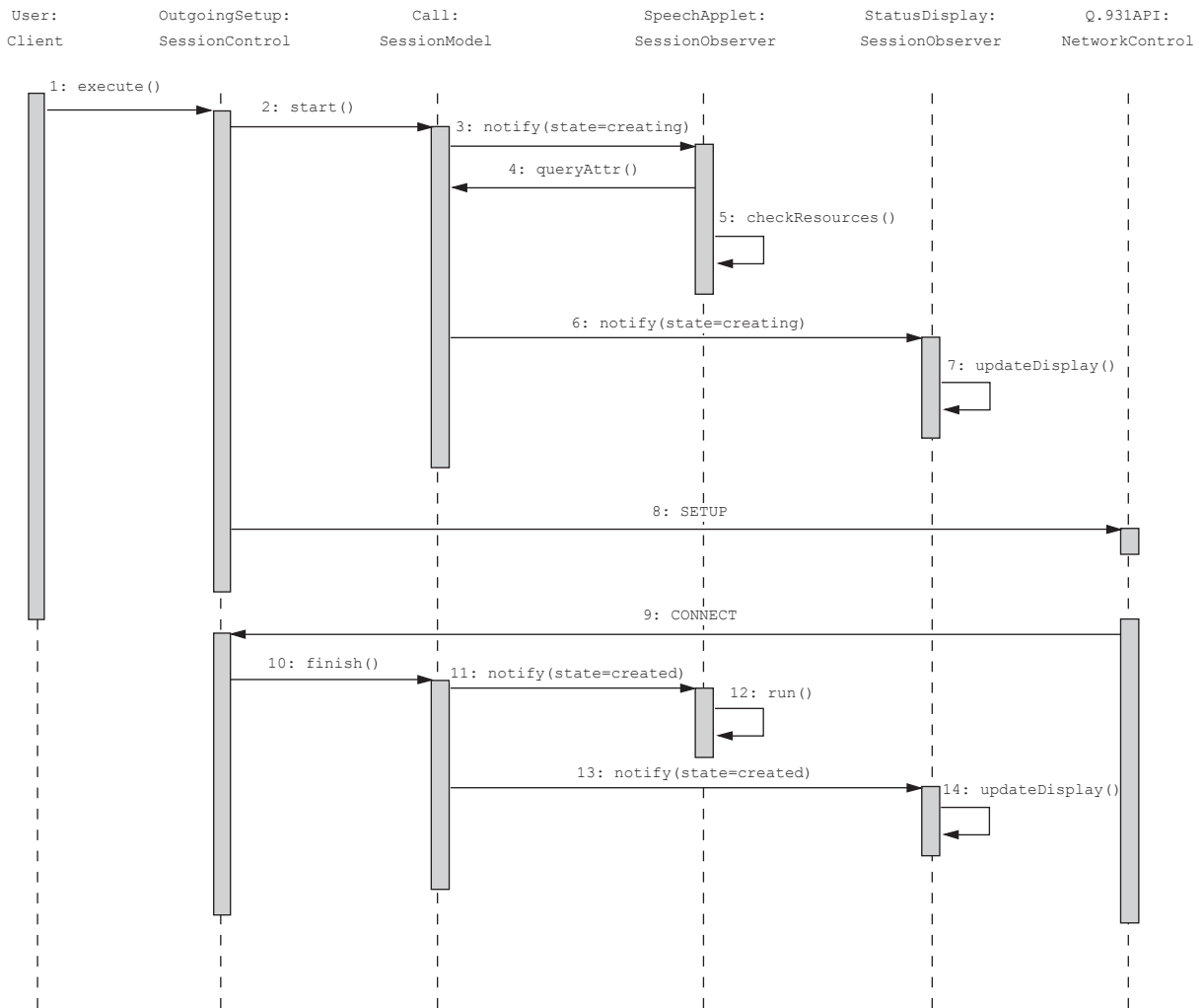
The pattern does not enforce a particular distribution of its participating classes. For example, `SessionControl` and `SessionModel` may reside on a central server, while clients may maintain `SessionObservers`. Notifications from `SessionModel` will be carried from the server to clients. Smart caching of `SessionModel` within clients may reduce overhead for `SessionModel` queries over the network.

See Also

Buschmann et al.¹⁰ describe several variants of the *Model View Controller* pattern.



(a)



(b)

Figure 7.
 (a) A class structure example of the Session Control & Observation pattern.
 (b) An application example of the Session Control & Observation pattern.

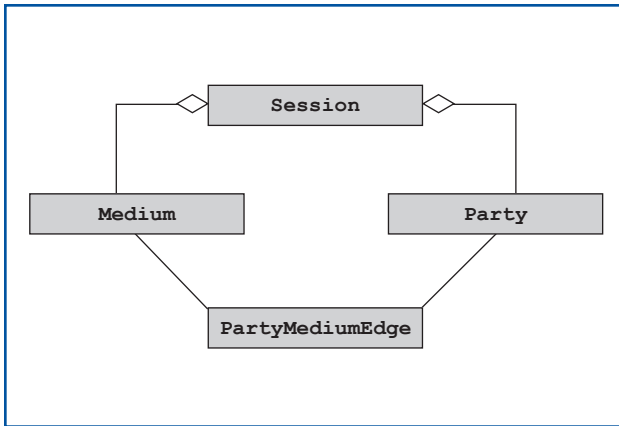


Figure 8.
A class structure example of the Parties & Media as First Class Citizens pattern.

Parties & media as first class citizens. The pattern *Parties & Media as First Class Citizens* was applied to shape the main structure of the *SessionModel* role within the *Session Control & Observation* pattern.

Name

Parties & Media as First Class Citizens.

Context

Clients conducting a session involving multiple parties and connections must maintain complex state information. As the session proceeds, this state is modified in various ways, such as adding new connections and removing parties. For example, clients of broadband ATM protocols, such as International Telecommunications Union (ITU) Q.2931,¹⁴ have to track parties and connections involved in a call.

The relationship between parties and connections (who is connected to whom) is often modeled from different viewpoints (that is, local view versus global view).

Problem

Different viewpoints often require complex conversions between models. How can we define and relate session entities that enable a single model to represent both a local and a global view?

Forces

- At times, parties in a session need to maintain connections.
- At times, connections need to maintain parties.
- A session release should clear all parties and connections.
- Services such as third-party call setup

require a global view at the initiator.

Solution

Have each session (*Session*) maintain parties (*Party*) and media (*Medium*) involved in the session. Use a third object (*PartyMediumEdge*) to maintain the state for each *Party* associated with *Medium*. Parties and media are thus first class citizens within a session. **Figure 8** shows this structure.

Medium embodies attributes common to all *Party* objects sharing it, such as agreed video coding. *PartyMediumEdge* represents aspects that are local to each *Party* with respect to that *Medium*, such as permissions and send/receive directivity. Another way of viewing the model is as a matrix of *Medium* and *Party* objects, where cells represent *PartyMediumEdge* objects.

Resulting Context

Any topology of parties and media can be represented. The session context can be modified by adding/deleting any combination of *Medium*, *Party*, and *PartyMediumEdge*. For example, a new *Party* can be added to an existing *Medium* by adding a *Party* and a *PartyMediumEdge*.

The pattern does not imply any particular distribution. Objects comprising a model may be distributed or centralized.

Examples

Figure 9 shows the session model developed for *MediaBuilder*. It applies the pattern twice, integrating a user-level context (*Session*, *Party*, *Medium*, *PartyMediumEdge*), as well as a communication context (*Association*, *Connection*, *PartyAssociationEdge*). The communication context can represent various (logical) connection topologies, such as point-to-point or multipoint.

The session management API presents the user/QoS portion of the model. See the *Facade* pattern, discussed below.

Known Uses

A similar modeling approach has been applied in related projects for broadband (ATM) signaling.^{15,16}

Facade. The *Facade* pattern, summarized here, is described in Gamma et al.⁷

Name

Facade.

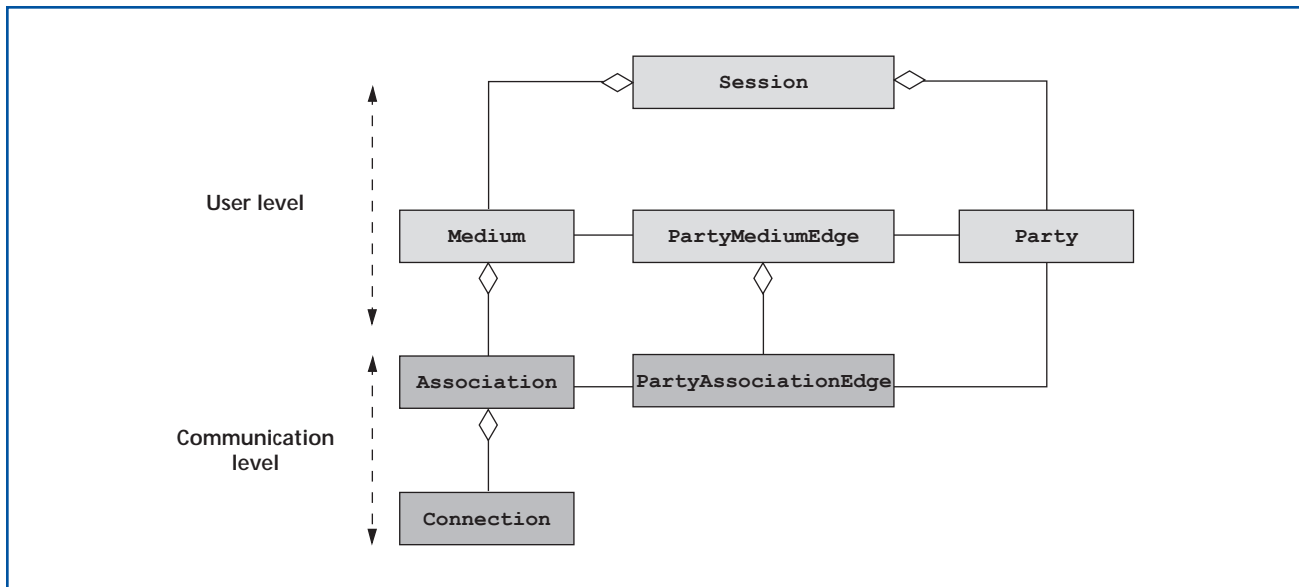


Figure 9.
A MediaBuilder session model.

Context

A system is structured into subsystems to reduce complexity.

Problem

How can coupling between subsystems be minimized?

Forces

High-level interfaces shield clients from the complexities of a subsystem, but they also hide lower-level functionality that clients with specialized needs may require.

Solution

Provide a high-level interface through a class (Facade) that shields clients from the lower-level functions of subsystem classes. Facade maps a high-level interface to low-level interfaces within a subsystem (see **Figure 10a**). Specialized clients may still access subsystem classes directly.

Resulting Context

- The pattern reduces the number of objects that clients have to deal with, making the subsystem easier to use.
- The pattern promotes weak coupling between subsystems and their clients, making the evolution of subsystems less likely to affect clients.
- A client can still access subsystem classes as needed.

Examples

Facade is applied in MediaBuilder to realize the session management API (see **Figure 10b**). The ServiceConvener class plays the Facade role, and SessionControlCommand is a subsystem class. ServiceConvener, which application clients invoke to modify the session, instantiates and invokes one or more SessionControlCommands to carry out the request.

See Also

Command Processor, described in Buschmann et al.,¹⁰ combines elements of Facade and Command.

Command. The Command pattern is applied to modify the session state in cooperation with peer session management entities. Command is described in Gamma et al.⁷ and summarized below.

Name

Command.

Context

A system to which clients can add new operations. These operations are invoked by the system and operate in a client-specific context. For example, we describe how to add a new type of signaling to MediaBuilder for performing session management.

Problem

How can the system invoke client-specific opera-

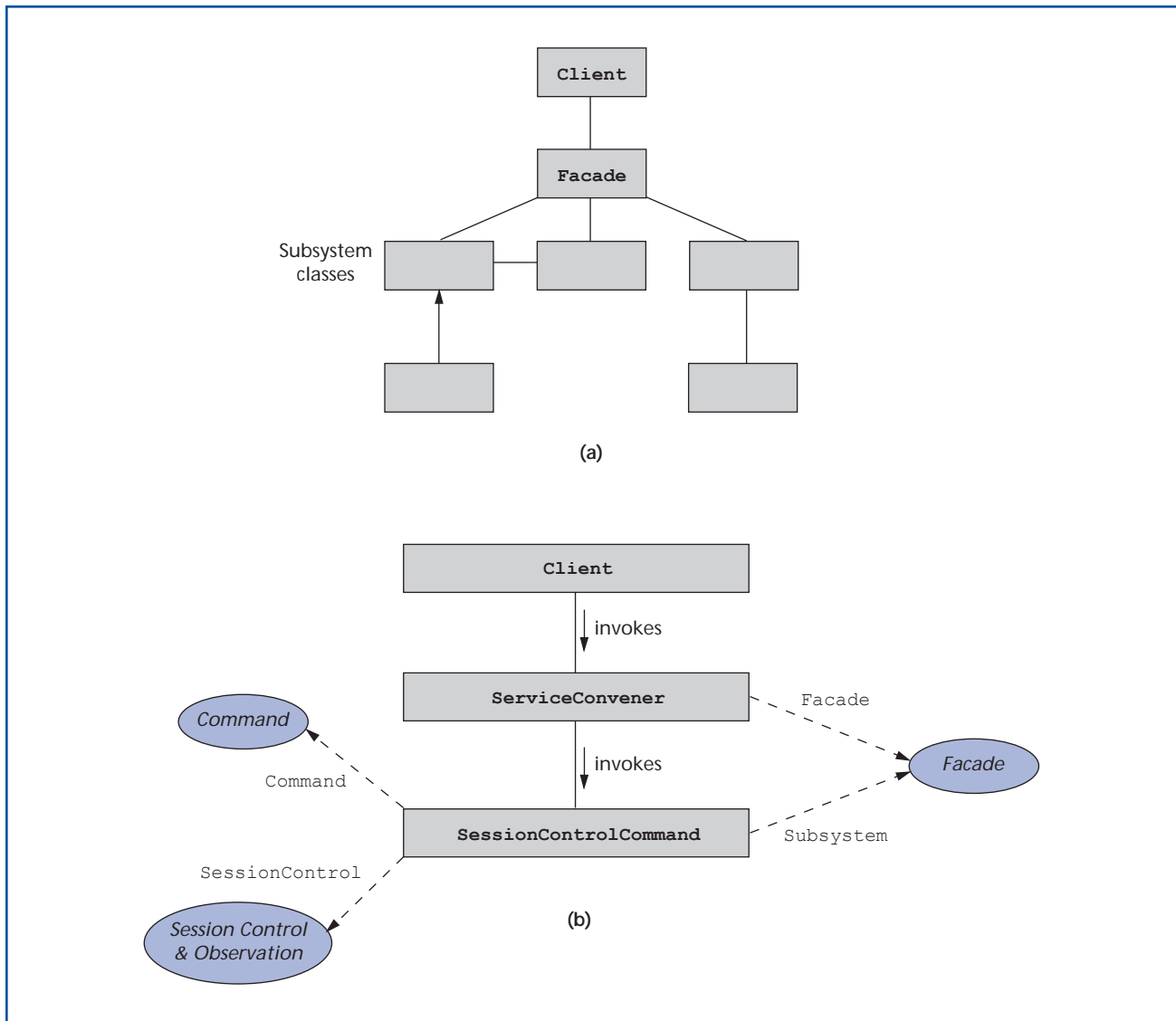


Figure 10. (a) The Facade pattern (see Gamma et al.,⁷ pp. 185-193). (b) Using the Facade pattern in MediaBuilder.

tions without knowing what they are or the context in which they will operate?

Forces

- The operation may be asynchronous, that is, the response to a request is not automatically returned.
- The client should be able to undo the operation before it is completed.

Solution

Encapsulate an operation as a class that declares an interface for its execution (Command). The actual execution is implemented in a derived

class (ConcreteCommand). For example, Command may declare a pure virtual member function—execute()—that is overloaded in ConcreteCommand. Similarly, the operation can be canceled by overloading a pure virtual undo() member function.

Resulting Context

Command decouples the system invoking the operation from the object that knows how to perform or undo it, in this case, ConcreteCommand.

Examples

MediaBuilder applies Command to realize session

management procedures (see Figure 10b). *SessionControlCommand* plays a role in two other patterns, *Session Control & Observation* (the *SessionControl* role) and *Facade* (the *Subsystem* role). *SessionControlCommand* can be specialized for different types of underlying networks and control architectures, such as central server or distributed. *ServiceConvener*, the invoker of the *SessionControlCommand*, does not need specific knowledge of these technologies. *ServiceConvener* uses the *Pluggable Factory* pattern (see “Application Engineering Patterns,” later in this paper) to create the appropriate *SessionControlCommand* and calls on its specialized `execute()` or `undo()` function. For example, within the PLATINUM project (see Panel 2), *SessionControlCommand* is specialized for ATM signaling procedures.

See Also

Command Processor, described in Buschmann et al.,¹⁰ combines elements of the *Facade* and *Command* patterns.

Multimedia Realization Patterns

Multimedia realization patterns address two issues:

- Modeling application protocols/streams and their relationships with multimedia devices and network transport media, using the *Layers* pattern; and
- Controlling the life cycle of application protocols/streams according to state changes in the session context, using the *Builder* pattern.

Layers. The *Layers* pattern is described in Buschmann et al.¹⁰ Although the pattern’s applicability is much wider, our summary is adapted to the domain of networking protocols.

Name

Layers.

Context

A system of networking protocols requires decomposition.

Problem

How should protocol components be modeled?

Forces

- Higher-level protocol components depend on lower-level ones.

- Components should be interchangeable.
- The highest and lowest protocol levels interact with system boundaries, such as applications and drivers.

Solution

Structure the system into an appropriate number of protocol layers, with the highest protocol layer on top and the lowest on the bottom. An actual design can have many variations (see “Examples,” below).

Examples

Within *MediaBuilder*, an application protocol stack has to interact at both ends with external boundaries. Multimedia devices are at the top, and network transport connections, such as native ATM or TCP/IP, are at the bottom. The stack itself may consist of several layers of protocols, but it may also be empty if a transport connection is directly coupled to a multimedia device (such as a native ATM stream coupled to a video window). We also need to cater to stack topologies, in which multimedia devices and transport media have a many to many relationship. For example, multiple incoming audio streams are sometimes mixed (called combining) and sent to both a speaker device and an audio file (known as splitting).

Figure 11 shows the basic class structure. A protocol stack consists of layered components (*StackElements*). Each component can be a “true” layer (*Layer*), such as a protocol, or can provide adaptation (*Adapter*) to the user level (*PresenterAdapter*) or the network level (*TransportAdapter*). Layers may be specialized to split (*Splitter*) or combine (*Combiner*) media streams. Other layers can implement protocols (*ProtocolLayer*, not shown).

See Also

Hüni et al.¹⁷ describe how GOF patterns such as *Strategy*, *Visitor*, and *Command* are applied in a framework for network protocols.

Builder. The *Builder* pattern, described in Gamma et al.,⁷ is summarized here.

Name

Builder.

Context

Construction of a composition of objects.

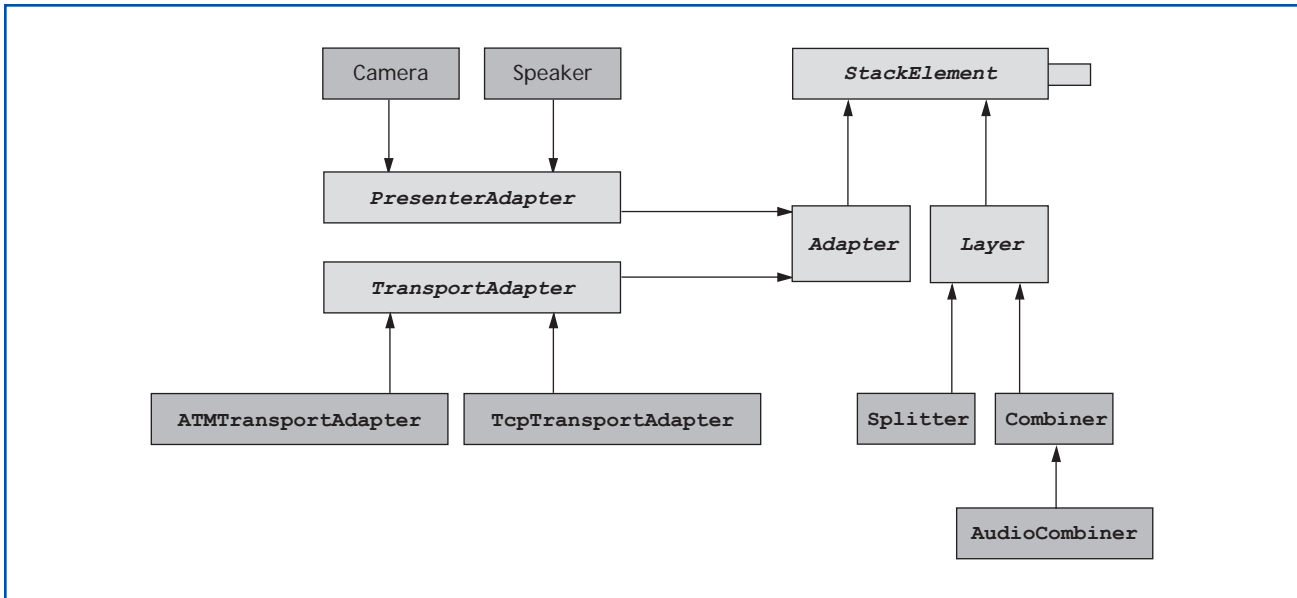


Figure 11.
An application of the Layers pattern in MediaBuilder.

Problem

How can we make the construction process and the composition transparent to the client?

Forces

The number of possible compositions is open ended.

Solution

Separate the construction of the composition from its representation to allow the same construction process to create different compositions. The construction process is provided by an abstract class, the Builder. The client instructs Builder to build a composition. By overloading Builder, the client can force it to create different compositions.

Examples

Within MediaBuilder, Builder is applied to create a concrete representation of the session context that is local to the application. This consists of a protocol stack with attached multimedia devices and transport connections (see the Layers pattern). The class MediumBuilder conforms to both Builder and the role of SessionObserver within the Session Control & Observation. It controls the life cycle of the protocol stack according to session state changes.

Application Engineering Patterns

The design patterns discussed in the sections

“Session Management Patterns” and “Multimedia Realization Patterns” are implemented as a set of base classes that together determine the basic behavior of the MediaBuilder framework. Extensions of these classes are specialized for technologies such as multimedia types or ATM signaling control. These extended classes, which form a layer of components on top of the abstract framework, are available for reuse. New applications can be created by selecting and combining components at a level higher than a programming language. This is sometimes called “programming without programming.” The Pluggable Factory pattern addresses these issues.

Name

Pluggable Factory.

Context

Provide reusable extensions of a (C++) framework. An object-oriented framework contains cooperating base classes that framework users can customize for their applications. The dilemma here is that the framework must instantiate these classes, even though it only knows about its base classes, which it cannot instantiate. The patterns Factory Method and Abstract Factory⁷ provide a solution based on abstract interfaces with signatures returning a base class object. Users can customize the implementation of these interfaces

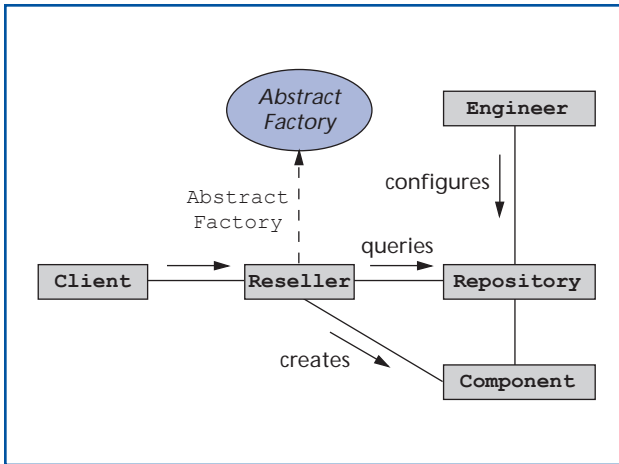


Figure 12.
An example of the Pluggable Factory pattern in MediaBuilder.

through inheritance such that an instance of their concrete class is returned. This type of mechanism can be regarded as an object-oriented form of application call-back. Many frameworks use *Factory*-type patterns. However, framework users still have to explicitly overload factory interfaces. Deriving new classes requires updating the factory and recompiling the application.

Problem

How can we make a factory whose product line can be configured by plugging in derived classes created by the framework? For example, if we introduce a fax medium into a MediaBuilder application, we need classes for a fax protocol and a fax multimedia device (see the *Layers* pattern). How can we add these to the application without recompiling or even stopping the application?

Solution

Extend the GOF factory patterns by symbolically creating objects from a repository (see **Figure 12**). Engineers (*Engineer*) configure component classes in a repository (*Repository*), each class of which has a symbolic name. When a client (*Client*) requests it, a reseller (*Reseller*) is able to instantiate a class from the repository through its symbolic name. Clients can also query the repository to get a list of class names. Implementing *Repository* works best with libraries that can be dynamically loaded, such as dynamic link libraries (DLLs) under Microsoft Windows.* Initialization files can store the class names and the specific library name that creates and implements them.

Resulting Context

Adding components to the repository can extend an application. The repository can also store symbolic relationships between components, enabling engineers to create applications by combining components.

Examples

Within MediaBuilder, *Pluggable Factory* provides a simple form of application engineering. By configuring class relationships in a file, it uses a repository of specialized classes to build larger components. At certain points, MediaBuilder will request *Pluggable Factory* for the creation of objects. For example, it may request that a voice connection be created by the base class *AudioSpeakerAdapter*. A user may have developed *MySpeakerAdapter* from *AudioSpeakerAdapter* to specify off-line that this class should be instantiated at run time.

Expanded Architecture

Figure 13 expands Figure 3 by showing key classes of the MediaBuilder architecture.

ServiceConvener, which provides the Session Management API, enables a user program to modify a session context by adding, deleting, and modifying session objects such as *Party* and *Medium*. *ServiceConvener* invokes specialized *SessionControlCommands* that handle the control procedures.

SessionControlCommand conforms to the *Command* pattern and the *SessionControl* role within the *Session Control & Observation* pattern. It is responsible for modifying the session state. *SessionControlCommand* objects are customized to use specific control architectures and network control APIs to carry out functions such as ATM signaling.

The state of the session is realized with the pattern *Parties & Media as First Class Citizens*. This context is shared by all MediaBuilder instances participating in that context. It constitutes a global view consisting of a graph of objects such as all *Parties*, *Media*, their associations (*PartyMediumEdges*), and the connection topology used to provide end-to-end transport flows. Only *SessionControlCommand* objects make changes to the session state. *SessionObserver* objects are “views” of the session context. They are notified when the session changes state and can act on

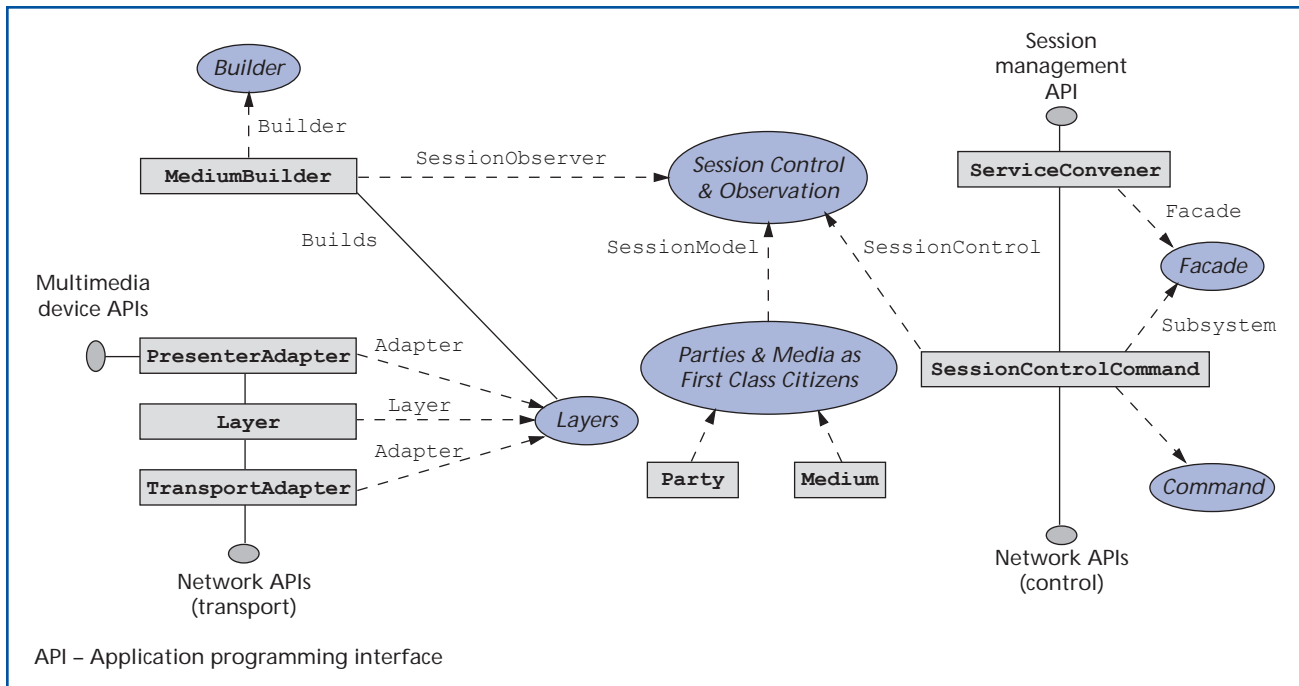


Figure 13. The MediaBuilder patterns architecture showing key classes.

these changes according to their roles (see the paragraph below).

The *MediumBuilder* is a *SessionObserver* that builds a concrete local representation of the session context using the structure of the *Layers* pattern. This representation comprises stacks of streams and protocols, as well as multimedia devices attached to them.

Layers integrates protocol layering (using *Layer* classes) and adaptation to boundaries (using *Adapter* classes) at the ends of a protocol stack. *TransportAdapters* map a specific transport interface, such as ATM or TCP/IP, to a common transport interface. *PresenterAdapters* work like multimedia devices. For example, a *CameraPresenterAdapter* maps video input streams from a camera device (driver) to the next lower element in the stack, which can be either a *Layer* or *TransportAdapter* object.

The *Reseller/Repository* classes (not shown in Figure 13) provide global services for configuring and instantiating components. The classes discussed above determine basic behavior, but the objects that carry out these responsibilities are created through the

Reseller. The framework calls on the *Reseller* to create objects to carry out specific tasks. Using the *Pluggable Factory* pattern, the *Reseller* creates specific objects that are configured from overloaded classes in a repository. For example, the *ServiceConvener* asks the *Reseller* to create an *OutgoingAddSessionControlCommand* when a user adds objects (such as a *Party*) to a session. Depending on how the repository was configured, the *Reseller* will return a specialized *OutgoingAddSessionControlCommand* (for example, for a specific type of ATM signaling). The repository also holds relationships between classes, such as the specific *SessionObservers* that are to be created for objects within the session model.

Application Example

The conference management (CM) application is a desktop tool for teleconferencing. It allows a user to share multiple media with one or more users. Several meetings can be held concurrently by a CM application. To manage these sessions, CM provides an easy-to-use control interface (see Figure 14) that visually reflects the session state (see “Parties & Media as First

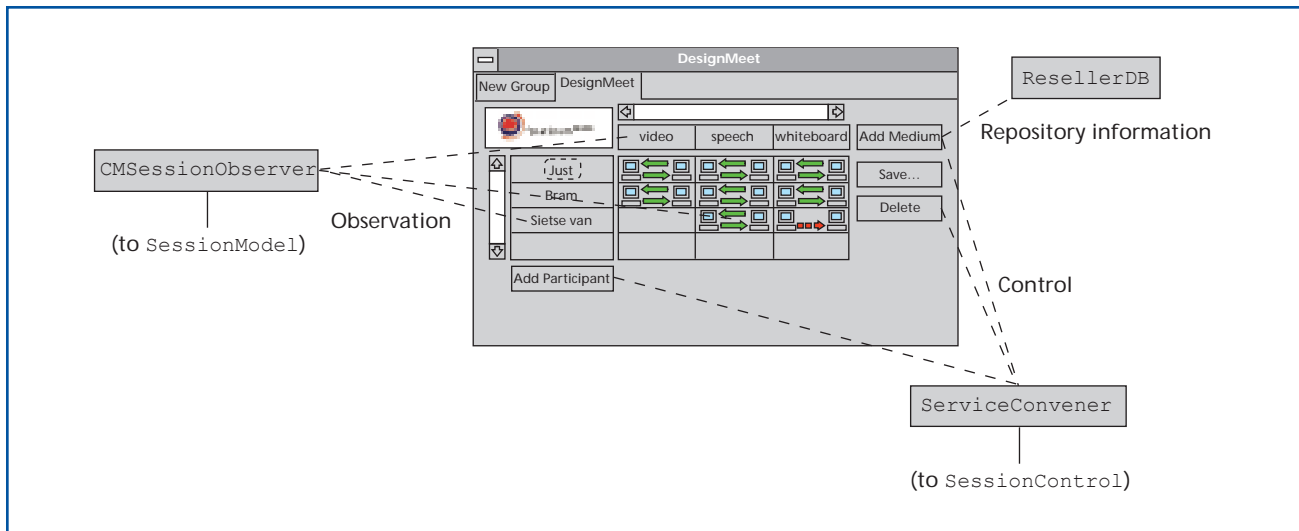


Figure 14.
The control interface for the conference management tool.

Class Citizens,” earlier in this paper). Vertically, it shows the participants in a group meeting, horizontally the media. Intersections show the participation status of each participant in a medium. For example, Just and Bram are participating in video, speech, and whiteboard, while Sietse is only participating in speech. Double arrows indicate that the participant both sends and receives on each medium. In addition, we see that Sietse is just being added to the whiteboard medium. This intermediate status is shown with a dashed arrow. The index-type tabs shown at the top of the screen in Figure 14 enable a user to switch between meetings.

Each participant can configure his or her own participation and media and that of other participants by using the “Add Participant” and “Add Medium” buttons. By clicking on the buttons of the participant/medium intersections, the participant can alter his or her own status and that of other participants. A user can also save an active meeting in a file and later return to the same meeting by loading that file. Participants can be given roles and permissions with respect to session management. These permissions are checked by the framework and reflected in the user interface, for example, by disabling particular buttons.

Only the graphical user interface (GUI) is specific to the CM application. All other functions are provided

by reusing MediaBuilder components. Networking functions are configured to use ATM signaling and native ATM streams (for optimal performance). The Medium Realization part reuses components like audio, video, and shared whiteboard. New multimedia components can be added (plugged in) without recompiling the application. The “Add Medium” button lets the user choose from a list of media available in the repository. When a medium is added to a session, its related protocols and multimedia devices are also automatically instantiated.

Figure 14 also shows interfaces to MediumBuilder. User actions that result in adding or deleting participants, media, or relationships between them map directly to the ServiceConvener. The CMSessionObserver is a SessionObserver customized for this application. It displays all active Parties, Media, and PartyMediumEdges within a session. **Figure 15** shows a picture of the entire application, including the multimedia realization components—video, whiteboard, and chat.

Conclusions

Patterns are a valuable tool to structure, communicate, and document the complex design issues that arise when combining networking and multimedia. Our experience with MediaBuilder showed us that designing with patterns was not a matter of “finding

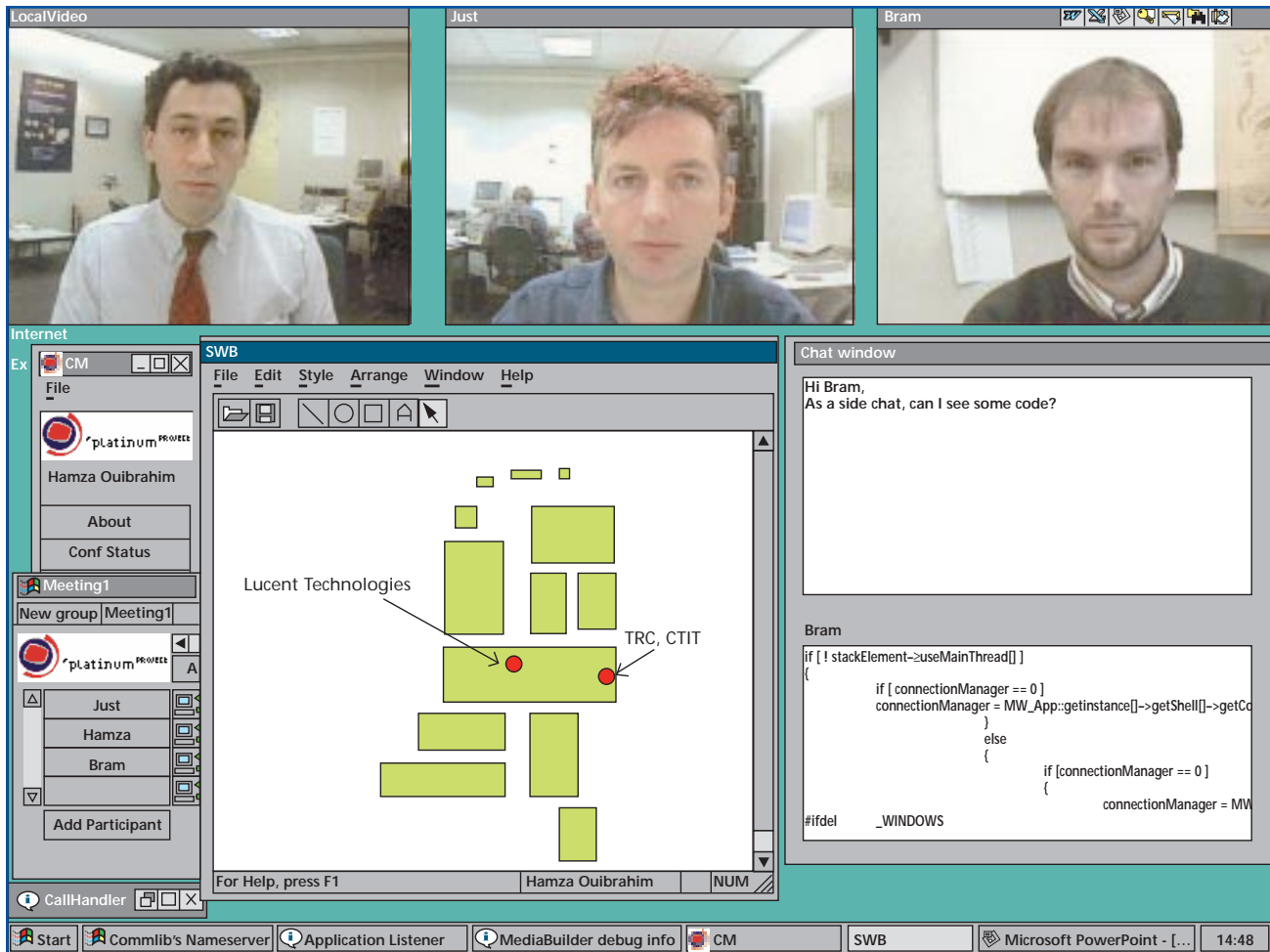


Figure 15. Using the conference management tool in a teleconference with audio, video, whiteboard, and chat.

all the patterns and proceeding with the implementation.” We started the MediaBuilder design with a single pattern (*Session Control & Observation*). During the course of the design, and even in the implementation, we discovered and applied additional patterns. Through iteration, the basic framework took shape. As the development team grew comfortable with patterns, they assimilated its language into their vocabulary. For example, one could hear phrases like: “Okay, let’s apply *Command* here.” The ability to communicate in abstract structures rather than detailed objects was another benefit of using patterns.

The patterns described in the GOF⁷ and POSA¹⁰ books are, in our opinion, a valuable starting point for projects and organizations that would like to apply design patterns.

Future Directions

We are continuing to develop MediaBuilder, particularly in these areas:

- A pilot project with users of teleteaching and remote medical consultation,
- Server applications,
- Additional desktop (client) applications,
- Support for Internet-related networking standards,
- Networking through Winsock2 provider modules, and
- Longer-term evolution to a CORBA/TINA-C architecture.^{4,18}

Acknowledgments

We thank all the reviewers of this paper—especially Doug McIlroy, Mark Bradac, Ferry van Geffen,

and Hamza Ouibrahim, all from Lucent Technologies—for their comments and helpful suggestions.

*Trademarks

Windows is a registered trademark of Microsoft Corporation.

Windows® 95 is a trademark of Microsoft Corporation.

Microsoft Windows NT is a trademark of Microsoft Corporation.

References

1. P. A. Bernstein, *Middleware—An Architecture for Distributed Systems*, White Paper CRL 93/6, Digital Equipment Corporation, Mar. 2, 1993.
2. A. A. Lazar, *Control, Management and Telemedia (COMET) Research Group—Activity Report*, Center for Telecommunications Research, Columbia University, New York City, Aug. 1996.
<http://www.ctr.columbia.edu/comet/activity-report/>
3. A. A. Lazar and K. S. Lim, "Programmability and Service Creation for Multimedia Networks," *Fifth IEEE International Symposium on High-Performance Distributed Computing*, Syracuse, New York, Aug. 1996, pp. 217-223.
4. Telecommunications Information Networking Architecture Consortium (TINA-C), *Overall Concepts and Principles of TINA*, TINA-C Deliverable version 1.0, 1995.
<http://www.tinac.com>
5. *Multimedia Communications Forum*,
<http://www.mmcf.org/>
6. K. Beck and R. E. Johnson, "Patterns Generate Architectures," *Proceedings of ECOOP94*, Bologna, Italy, July 1994, pp. 139-149.
<http://st-www.cs.uiuc.edu/users/patterns/papers>
7. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns—Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts, 1995.
8. G. Meszaros, "A Pattern Language for Improving Capacity of Reactive Systems," *Pattern Languages of Program Design—2*, ed. J. Vlissides et al., Addison-Wesley, Reading, Massachusetts, 1996, pp. 575-592.
9. M. Adams et al., "Fault-Tolerant Telecommunication System Patterns," *Pattern Languages of Program Design—2*, ed. J. Vlissides et al., Addison-Wesley, Reading, Massachusetts, 1996, pp. 549-562.
10. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture—A System of Patterns*, John Wiley, New York, 1996.
11. E. Gamma, *Object-Oriented Software Development based on ET++*, *Design Patterns, Class Library, Tools* (in German), Springer-Verlag, Berlin, 1992.
12. G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Method for Object-Oriented Development*, Documentation Set 0.91, Rational Software Corporation, Sept. 1995.
<http://www.rational.com/ot/uml>
13. G. E. Krasner and S. T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in SmallTalk-80," *Journal of Object-Oriented Programming*, Vol. 1, No. 3, Aug. 1988, pp. 26-49.
14. ITU Q.2931, *Broadband Integrated Services Digital Network (B-ISDN)—Digital Subscriber Signaling No. 2 (DSS 2). User Network Interface Layer 3 Specification for Basic Call/Connection Control*, International Telecommunications Union, 1995.
15. *Service Description Framework and B-ISDN Service Descriptions*, RACE II/MAGIC Deliverable 3, June 1993.
<http://www.analysys.com/race>
16. S. Minzer, "A Signaling Protocol for Complex Multimedia Services," *IEEE Journal on Selected Areas in Communications*, Vol. 9, No. 9, Dec. 1991, pp. 97-114.
17. H. Hüni, R. E. Johnson, and R. Engel, "A Framework for Network Protocol Software," *OOPSLA '95 Proceedings, ACM SIGPLAN Notices*, Austin, Texas, Oct. 1995, pp. 358-369.
18. *Object Management Group*, <http://www.omg.org/>
<http://st-www.cs.uiuc.edu/users/patterns/papers>
19. A. Klapwijk, H. Ouibrahim, and U. Behnke, "PLATINUM: A Platform for New Users of Multimedia," *European Conference on Networks & Optical Communications (NOC'96)*, Heidelberg, Germany, June 1996, pp. 115-122.
20. H. Ouibrahim and J. A. van den Broecke, "Multiparty/Multimedia Services to Native ATM Desktops," *Fifth IEEE International Symposium on High-Performance Distributed Computing*, Syracuse, New York, Aug. 1996, pp. 203-208.
21. G. Booch, *Object-Oriented Analysis and Design with Applications*, 2d ed., Benjamin/Cummings, Redwood City, California, 1993.
22. J. Rumbaugh, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
23. I. Jacobson, *Object-Oriented Software Engineering—A Use Case Driven Approach*, Addison-Wesley, Workingham, U.K., 1992.
24. G. Booch, "Patterns and Protocols," *Report on Object-Oriented Analysis and Design (ROAD)*, Vol. 2, No. 3, May-June 1996, pp. 2-8.

Further Reading

J. O. Coplien, "Pattern Languages of Program

Design,” ed. D. C. Schmidt, *First Annual Conference of Pattern Languages of Programs (PLoP)*, Monticello, Illinois, Aug. 1994.

J. O. Coplien, *Software Patterns*, SIGS Publications, New York, June 1996.

R. P. Gabriel, *Patterns of Software: Tales from the Software Community*, Addison-Wesley, Oxford, U.K., Aug. 1996.

(Manuscript approved February 1997)

JUST A. VAN DEN BROECKE, who recently left the company, was a member of technical staff in the Forward Looking Work Department of Network Systems in Huizen, The Netherlands, when this work was performed. He was responsible for the software architecture of the middleware for advanced telecommunications services based on CORBA/TINA-C standards. His research interests include distributed object technology, especially CORBA; multimedia networking; and network management. Mr. van den Broecke received an Ms.C. in chemistry and computing science from the University of Amsterdam, The Netherlands.



JAMES O. COPLIEN is a principal investigator in the Software Production Research Department at Bell Laboratories in Naperville, Illinois. He carries out research programs in software design patterns, empirical organizational modeling, multiparadigm design, and the object paradigm; he is also a C++ expert. Mr. Coplien earned a B.S. in electrical and computer engineering and an M.S. in computer science from the University of Wisconsin at Madison. ♦

