


Catalysts

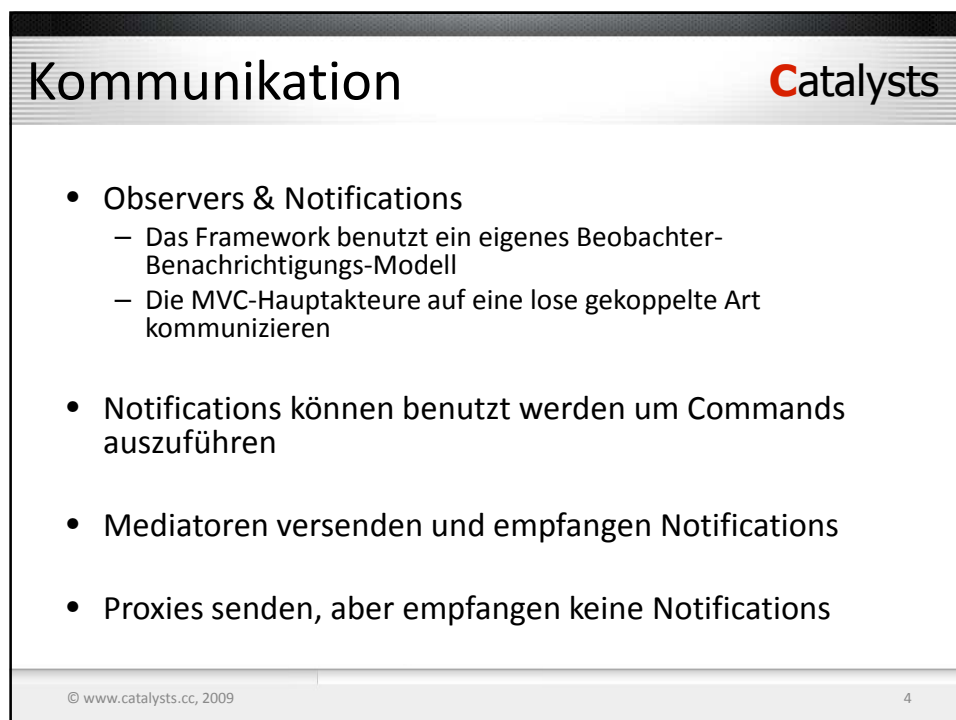
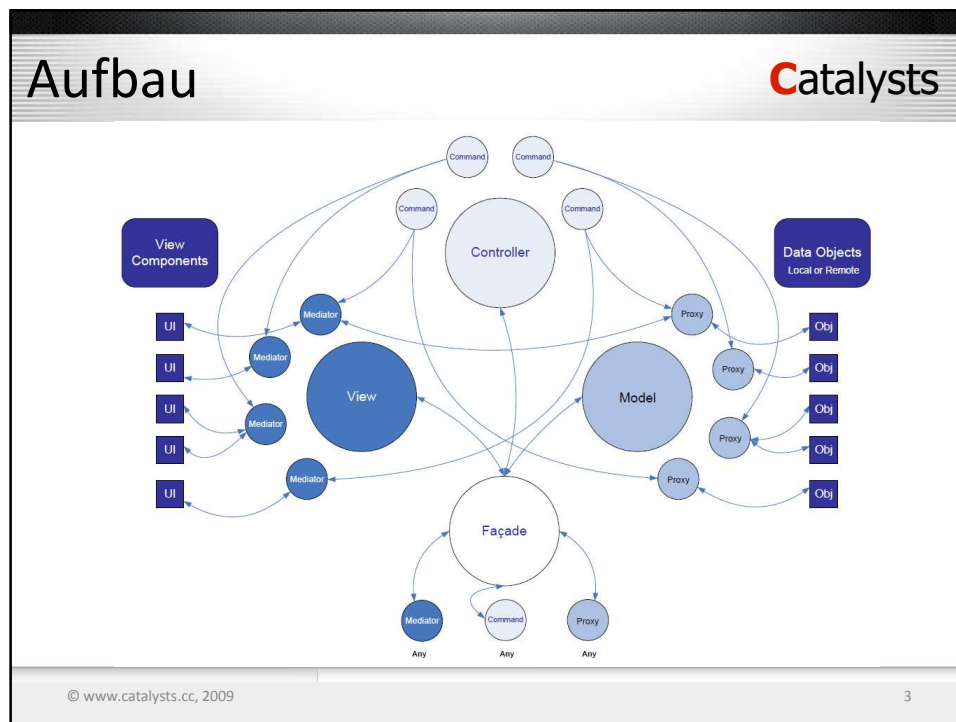
pure  mvc

DI Harald Radi  
Catalysts GmbH

Übersicht Catalysts

- Das PureMVC Framework bietet eine Hilfe um das Model – View – Controller Architekturmuster in Softwareprojekten umzusetzen
- PureMVC entstand aus der Notwendigkeit leistungsstarke RIA Anwendungen entwickeln zu können
  - Referenzimplementierung in Actionscript
  - Mittlerweile in fast jeder Programmiersprache verfügbar

© www.catalysts.cc, 2009 2



# Kommunikation Catalysts

- Mediator
  - horcht auf Ereignisse die in der View auftreten
  - versendet Nachrichten um Commands anzustossen
- Command
  - registriert sich für einen bestimmten Nachrichtentyp
  - beinhaltet Logik um Nachrichten „abzuarbeiten“
  - Referenziert Proxies um das Model beeinflussen zu können
- Proxy
  - verwaltet Daten
  - interagiert mit Remote-Services
  - versendet Nachrichten um die View über Datenänderungen zu notifizieren oder weitere Commands anzustossen

© www.catalysts.cc, 2009 5

# Events vs. Notifications Catalysts

- Events
  - Um Events zu erhalten muss man sich beim Dispatcher registrieren, d.h. der Empfänger muss vorab schon den Absender kennen
  - In manchen Umgebungen können Events durch die Komponenten-Hierarchie „bubblen“, d.h. man kann sich für das „Click“-Event eines Buttons auch am beinhaltenden Dialog registrieren
- Notifications
  - Nachrichten werden über die Facade versendet, man kann daher auch der Facade die Interesse an einem bestimmten Nachrichtentyp mitteilen
  - Sender und Empfänger sind über die Facade lose gekoppelt
  - Es kann auch unterschiedliche Absender für einen Nachrichtentyp geben

© www.catalysts.cc, 2009 6

# Facade Catalysts

- Die Facade vermittelt Anfragen zu Model, View und Controller
- Die Klassen der Hauptakteure werden nie direkt benutzt
  - Auf Model, View und Controller wird im Anwendungscode nie direkt zugegriffen
  - Die Facade instanziiert diese Klassen automatisch
- Üblicherweise gibt es in jeder PureMVC Anwendung eine konkrete Facadenimplementierung
  - Die Facade initialisiert Command Mappings
  - Model und View Komponenten werden über Commands erzeugt

© www.catalysts.cc, 2009 7

# Facade Catalysts

```
// A concrete Facade for MyApp
public class ApplicationFacade extends Façade implements IFacade
{
    // Define Notification name constants
    public static const STARTUP:String = "startup";
    public static const LOGIN:String = "login";

    // Singleton ApplicationFacade Factory Method
    public static function getInstance(): ApplicationFacade
    {
        if ( instance == null ) instance = new ApplicationFacade( );
        return instance as ApplicationFacade;
    }

    // Register Commands with the Controller
    override protected function initializeController(): void
    {
        super.initializeController();
        registerCommand( STARTUP, StartupCommand );
        registerCommand( LOGIN, LoginCommand );
        registerCommand( LoginProxy.LOGIN_SUCCESS, GetPrefsCommand );
    }

    // Startup the PureMVC apparatus, passing in a reference to the application
    public function startup( app:MyApp ): void
    {
        sendNotification( STARTUP, app );
    }
}
```

© www.catalysts.cc, 2009 8

## Commands

Catalysts

- Commands werden in der Facade registriert
  - Facade erzeugt Controller welcher sich die Command Mappings merkt
  - Controller registriert sich bei der Facade als Empfänger für alle Nachrichtentypen zu denen Commands registriert sind
- Commands sind zustandslos
  - Commands werden vom Controller on-demand instanziiert
  - Beim Eintreffen einer Nachricht ruft der Controller die „execute“ Methode des zugehörigen Commands auf
  - Commands können daher auch keine Referenzen auf länger-lebende Objekte halten
- Simple- und MacroCommands
  - Nur ein Command pro Nachrichtenart registrierbar
  - Über MacroCommands sind Command-Batches möglich

© www.catalysts.cc, 2009 9

## Commands

Catalysts

- Commands können ...
  - bestehende Registrierungen von Mediatoren, Proxies und Commands überprüfen, ändern, hinzufügen und entfernen
  - Nachrichten versenden
  - Proxies und Mediatoren abfragen und manipulieren
- Daten können in verschiedene Teile der View transportiert werden
- Über Commands ist die Interaktion mit dem Model möglich

© www.catalysts.cc, 2009 10

# Commands Catalysts

```

// A MacroCommand executed when the application starts.
public class StartupCommand extends MacroCommand
{
    // Initialize the MacroCommand by adding its subcommands.
    override protected function initializeMacroCommand() : void
    {
        addSubCommand( ModelPrepCommand );
        addSubCommand( ViewPrepCommand );
    }
}

// Create and register Proxies with the Model.
public class ModelPrepCommand extends SimpleCommand
{
    // Called by the MacroCommand
    override public function execute( note : INotification ) : void
    {
        facade.registerProxy( new SearchProxy() );
        facade.registerProxy( new PrefsProxy() );
        facade.registerProxy( new UsersProxy() );
    }
}

// Create and register Mediators with the View.
public class ViewPrepCommand extends SimpleCommand
{
    override public function execute( note : INotification ) : void
    {
        var app:MyApp = note.getBody() as MyApp;
        facade.registerMediator( new ApplicationMediator( app ) );
    }
}

```

© www.catalysts.cc, 2009 11

# Mediatoren Catalysts

- Vermitteln zwischen Benutzer-Interaktionen über UI Komponenten und dem Rest der PureMVC Anwendung
  - Ein Mediator kennt seine View-Komponente vorab
  - und kann sich daher als Event Handler bei seiner View Komponente registrieren
- Ein Mediator entkoppelt die View-Komponente (das Design) gänzlich von der dahinterliegenden Anwendungslogik
- Mediatoren können ...
  - Nachrichten versenden und empfangen
  - Proxies abfragen und benutzen
  - Sub-Mediatoren anlegen

© www.catalysts.cc, 2009 12

# Mediatoren

Catalysts

## Faustregeln

- Sollen mehrere Mediatoren auf ein Benutzeraktion reagieren, so ist von dem der View-Komponente zugehörigen Mediator ein Proxy zu aktualisieren oder eine Nachricht zu senden – die weiteren Mediatoren können dann darauf reagieren
- Ist eine Menge an koordinierten Interaktionen mit anderen Mediatoren nötig, so sollte dieser Code stattdessen in einen Command ausgelagert werden
- Andere Mediatoren sollen nie direkt angesprochen werden; Mediatoren sollten nicht dahingehend erweitert werden dass von außen auf sie eingewirkt werden kann
- Behandelt ein Mediator mehr als 4-5 Nachrichtentypen, so sollte die Zuständigkeit auf mehrere Mediatoren aufgeteilt werden; statt für einen gesamten Dialog einen gigantischen Mediator zu verwenden, sollten eigene Mediatoren für abgrenzbare Bereiche des Dialoges erstellt werden

© www.catalysts.cc, 2009 13

# Mediators

Catalysts

```

// A Mediator for interacting with the LoginPanel component.
public class LoginPanelMediator extends Mediator implements IMediator
{
    public static const NAME:String = "LoginPanelMediator";

    public function LoginPanelMediator( viewComponent:LoginPanel )
    {
        super( NAME, viewComponent );
        LoginPanel.addEventListener( LoginPanel.TRY_LOGIN, onTryLogin );
    }

    // List Notification Interests
    override public function listNotificationInterests( ): Array
    {
        return [
            LoginProxy.LOGIN_FAILED,
            LoginProxy.LOGIN_SUCCESS
        ];
    }

    // Handle Notifications
    override public function handleNotification( note:INotification ):void
    {
        switch ( note.getName() ) {
            case LoginProxy.LOGIN_FAILED:
                LoginPanel.loginVO = new LoginVO();
                LoginPanel.loginStatus = LoginPanel.NOT_LOGGED_IN;
                break;
            case LoginProxy.LOGIN_SUCCESS:
                LoginPanel.loginStatus = LoginPanel.LOGGED_IN;
                break;
        }
    }
}

// User clicked Login Button; try to log in
private function onTryLogin ( event:Event ) : void {
    sendNotification( ApplicationFacade.LOGIN, loginPanel.loginVO );
}

// Cast the viewComponent to its actual type.
protected function get loginPanel(): LoginPanel {
    return viewComponent as LoginPanel;
}

```

© www.catalysts.cc, 2009 14

## Proxies

Catalysts

- Ein Proxy ist ein Platzhalter für ein Objekt auf das kontrolliert zugegriffen werden kann
- Durch die Verwendung eines Proxies wird der Ort der Datenhaltung abstrahiert
  - lokal verwaltete Daten
  - über ein Remote-Service verwaltete Daten
  - persistente Daten
- Ein Proxy kann nur Nachrichten versenden, nicht aber auf Nachrichten reagieren
  - nur die verwalteten Daten können von einem Proxy modifiziert werden
  - Zugriffe auf Mediatoren oder andere View-Komponenten sind untersagt
  - Änderungen an den Daten werden über Nachrichten bekanntgegeben

© www.catalysts.cc, 2009 15

## Proxies

Catalysts

- **Remote Proxy**
  - Ein konkreter Proxy, der serverseitige Daten verwaltet. Auf diese Daten wird über irgendeine Art von Service zugegriffen
- **Proxy und Delegate**
  - Mehrere Proxies benötigen den Zugriff auf das gleiche Service Object. Die Delegate Klasse verwaltet dieses Service Object und kontrolliert den Zugriff sowie die Weiterleitung der Ergebnisse an die Anforderer
- **Protection Proxy**
  - Wird benutzt, falls ein Objekt verschiedene Zugriffsrechte benötigt
- **Virtual Proxy**
  - Erstellt bei Bedarf ein 'teueres' Objekt
- **Smart Proxy**
  - Lädt beim ersten Zugriff das Data Object in den Speicher

© www.catalysts.cc, 2009 16

# Proxies Catalysts

```

// A proxy to log the user in
public class LoginProxy extends Proxy implements IProxy {
    public static const NAME:String = "LoginProxy";
    public static const LOGIN_SUCCESS:String = "loginSuccess";
    public static const LOGIN_FAILED:String = "loginFailed";
    public static const LOGGED_OUT:String = "loggedOut";
    private var loginService: RemoteObject;

    public function LoginProxy () {
        super( NAME, new LoginVO ());
        loginService = new RemoteObject();
        loginService.source = "LoginService";
        loginService.destination = "GenericDestination";
        loginService.addEventListener( FaultEvent.FAULT, onFault );
        loginService.login.addEventListener( ResultEvent.RESULT, onResult );
    }

    // Cast data object with implicit getter
    public function get loginVO(): LoginVO {
        return data as LoginVO;
    }

    // The user is logged in if the login VO contains an auth token
    public function get loggedIn(): Boolean {
        return ( authToken != null );
    }

    // Subsequent calls to services after login must include the auth token
    public function get authToken(): String {
        return loginVO.authToken;
    }
}

// Set the users credentials and log in, or log out and try again
public login( tryLogin: LoginVO ): void {
    if ( ! loggedIn ) {
        loginVO.username = tryLogin.username;
        loginVO.password = tryLogin.password;
    } else {
        logout();
        login( tryLogin );
    }
}

// To log out, simply clear the LoginVO
public function logout(): void {
    if ( loggedIn ) loginVO = new LoginVO();
    sendNotification( LOGGED_OUT );
}

// Notify the system of a login success
private function onResult( event: ResultEvent ): void {
    setData( event.result ); // immediately available as loginVO
    sendNotification( LOGIN_SUCCESS, authToken );
}

// Notify the system of a login fault
private function onFault( event: FaultEvent ): void {
    sendNotification( LOGIN_FAILED, event.fault.faultString );
}

```

© www.catalysts.cc, 2009 17

# Proxy Interaction Catalysts

```

public class GetPrefsCommand extends SimpleCommand {
    override public function execute( note: INotification ): void {
        var authToken: String = note.getBody() as String;
        var prefsProxy: PrefsProxy;
        prefsProxy = facade.retrieveProxy( PrefsProxy.NAME ) as PrefsProxy;
        prefsProxy.getPrefs( authToken );
    }
}

public class LoginCommand extends SimpleCommand {
    override public function execute( note: INotification ): void {
        var loginVO: LoginVO = note.getBody() as LoginVO;
        var loginProxy: LoginProxy;
        loginProxy = facade.retrieveProxy( LoginProxy.NAME ) as LoginProxy;
        loginProxy.login( loginVO );
    }
}

```

© www.catalysts.cc, 2009 18

## Zusammenfassung

Catalysts

- **MVC Architekturmuster:** Trennung von Präsentations-, Business- und Domain-Logik
- Das **PureMVC** Framework bietet Hilfsmittel um dieses Architekturmuster umzusetzen
- Mediators: beinhalten Präsentationslogik
- Commands: beinhalten Businesslogik
- Proxies: beinhalten Domainlogik