

Proceedings of ForUse 2003

[Session W22]

Lean Interaction Design and Implementation: Using Statecharts with Feature Driven Development

David J. Anderson, Brían O'Byrne

Abstract

Lean UI development in Feature Driven Development is achieved through right-first-time implementation of the interaction designer's intent using David Harel's Statechart notation to model the interaction design. Statechart notation can be directly mapped to an MVC Type-II architecture and Mediator Pattern. With a few minor extensions Statechart notation can be used to model complex application behavior such as exception handling and transactions. Statecharts can be mapped into UI Features for tracking using an FDD Knowledge Base. The Statechart can be drawn using a UML modeling tool, and imported into the JStateMachine engine which implements a table driven Mediator and Command pattern system in either a Web Servlet or JFC (Swing) environment. The result is reduced variation in UI development and precise implementation of the interaction designer's intent. JStateMachine (an application framework) validates the runtime code against the original interaction model insuring accurate runtime implementation of the UI design.

Introduction

Feature Driven Development (FDD) is one of the family of Agile software development methods. Some Agile methods have been criticized for not encouraging good usage-centered design [Constantine 2001]. Unlike some other Agile methods FDD had a role-oriented task usage design approach included in the original process defined in 1998. Since then some refinements in the technique have been introduced. The most important of these was the adoption of Statecharts for interaction design modeling following the publication of his book on the topic by Ian Horrocks [1999].

Also in the same year the JStateMachine [O'Byrne 2003] engine was introduced as a runtime environment for implementation of user interface designs modeled using Statecharts. The JStateMachine engine controlled and monitored the flow of logic in the user interface presentation layer code. Code executing as designed was allowed to run correctly, code producing unexpected events or unexpected navigation requests was elegantly isolated and gracefully handled through exception handling code which handed off to a suitable error message screen.

The introduction of Statechart modeling brought a clean, well-defined rigor to the definition of a user interface and the implementation in the JStateMachine engine brought reliable execution and enabled a table-driven, soft-coded approach for making late modifications to the design and implementation.

The combination enabled many aspects of Agile software development. Measurement of progress was achieved through working code coverage of the Statechart interaction design whilst late changes could be accommodated easily through modification of the Statechart and the regression impact of the change identified and isolated to minimize the refactoring effort with a resultant improvement in software quality.

Interest in Statechart modeling for interaction design has been growing steadily over the intervening four years and the recent popularity of presentation layer frameworks such as Struts has generated increasing interest in the JStateMachine engine.

Part1: Producing the Design with FDD

FDD is a lightweight development process. It is designed to be easy to learn and easy to execute. There are five stages to FDD as shown in Figure 1. The first three represent an initial architecture and planning spike, whilst the latter two represent the consistent, repeatable production of high quality working code.

Five collaborating processes

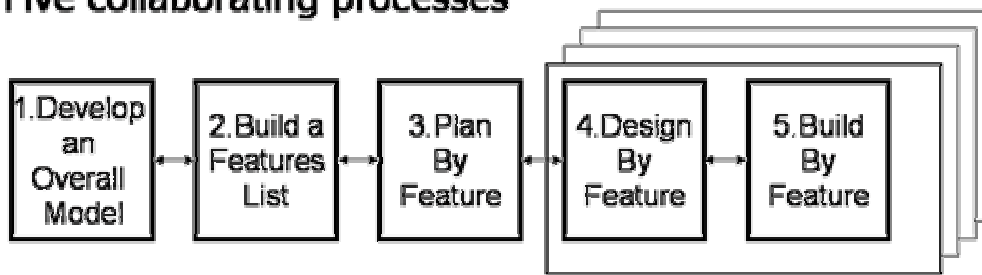


Figure 1 – Five Process in FDD

The original FDD process introduced process methodology not only for development of code but also for the development of the user interface design.

Develop an Overall Model

The purpose of the first stage in FDD is to develop a shared understanding of the work to be accomplished. This is done through a modeling exercise which is designed to be wide rather than deep – to cover the whole of the scope but in a fashion which produces a thin framework without much detail. For business logic and data persistence this is achieved using the Archetypes and Domain Neutral Component techniques of Peter Coad and colleagues [Coad 1999].

The challenge was to find a suitable equivalent for user interface. The answer was provided by Horrocks' work with Statecharts. Statecharts were adopted into David Anderson's work with FDD in Ireland in 1999. A Statechart representing the interaction (or navigation) model for the application became the output from step 1 of FDD for user interface.

Build a Features List

Jeff De Luca says “Features are tiny”. Peter Coad describes them as “small pieces of client-valued functionality”. Statecharts provide a convenient method for identifying such small pieces of client-valued functionality in the application. For every screen that can be viewed, there is a state, and for every action that can take place, there is a state transition triggered by an event. Therefore, it seemed only natural to list each viewable screen as a Feature and each actionable event, or navigation, as a Feature. Hence, the Feature List consists of each View and Event from the interaction design.

Such a list of “client-valued functionality” meets with the Agile principle of measuring delivered value. A project plan which tracks the delivery of the Features is a plan which is tracking delivery of value for the customer.

Plan by Feature

In the planning stage, Features are grouped into logical sets known generically as Feature Sets. Business logic (or Problem Domain - PD) Feature Sets are referred to as Business Activities because they reflect functionality which relates to a single line of business. UI Features are grouped into sets based on containers within the interaction design. A container usually bounds a logical group of activities which are saved, stored or persisted together, they may relate to tasks from a single role, or tasks which are logically performed together, in sequence or approximately at the same time – they have a temporal affinity. There are many reasons for a user interface designer to decide to group a set of tasks together within the same container element of a UI design.

Aggregation of Features produces a reduction in uncertainty due to the mathematical effect of aggregating the individual variance in size and complexity across Features. This effect, known as the square root of the sum of the squares, is well known in statistics and is explained with respect to design activity in Donald Reinertsen’s excellent book, “Managing the Design Factory” [Reinertsen 1997]. Reducing uncertainty makes planning more accurate and reduces the size of any buffer required for a whole project.

Dependencies between Feature Sets are then identified. Often UI Containers are dependent upon problem domain code for a Business Activity. There may also be dependencies between UI Features Sets or between Business Activities. A multi-threaded plan showing sequential dependencies is constructed, as shown in Figure 2. This is done using the

Critical Chain Project Management technique of Eli Goldratt [Goldratt 1997] which is better described in book of the same name by Lawrence Leach [Leach 2000].

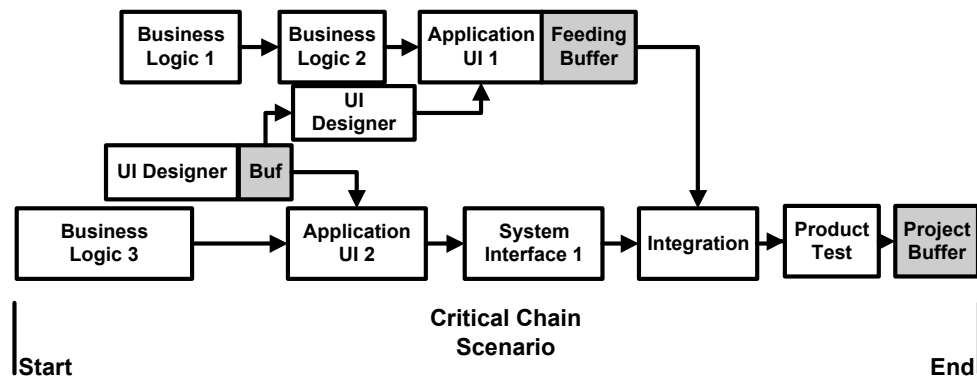


Figure 2 – Example Critical Chain plan for an FDD project

Features grouped into sets and subject areas are tracked using a knowledge base web tool. Figure 3 shows a screen capture from such a tool. A UI container called the Content Browser contains 7 Features – 2 Views and 5 Event handlers (Controllers).

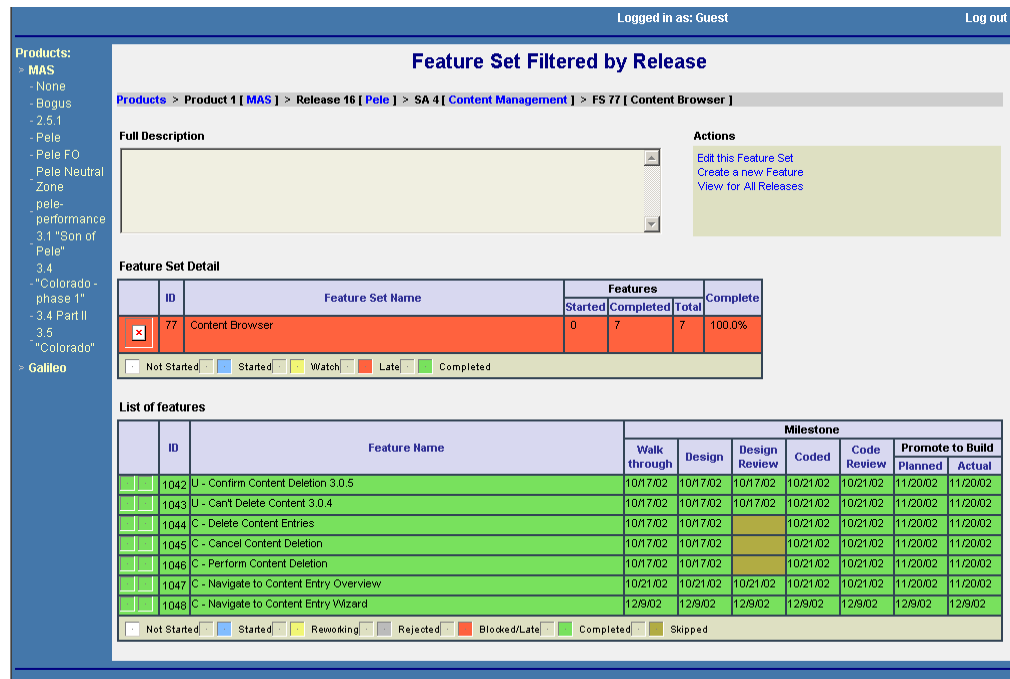


Figure 3 – Example UI Features grouped as a Feature Set

Basic Elements of a Statechart

Figure 4 shows all the basic elements of the Statechart notation. Statechart notation was created by David Harel [Harel 1987] as a technique to reduce the complexity in State Transition Diagrams. Statecharts are essentially hierarchical fractal representation of State Transition Diagrams. The introduction of containment of a sub-state within a super-state greatly simplifies the notation for modeling states and their transitions. It is this hierarchical containment which makes them appropriate for modeling user interfaces.

In Horrocks application of Statecharts to user interface interaction modeling, a state represents a steady state of the user interface – a view or screen shown to the user which requires a user action before transitioning to the next view or screen. The transitions between states are modeled with transition arrows. Each transition arrow is annotated with an event (or action) name, a list of data parameters, a guard statement which must be true for the transition to take place and an action which should be performed before the transition is completed. On entry, a state can begin from its starting sub-state, or from its most recent sub-state, shown on the

diagram with an H, or from a recursively derived sub-state, shown on the diagram with an H*. These history states allow modeling of typically useful UI behavior such as Cancel actions which result in a return to a previous state of the user interface and application.

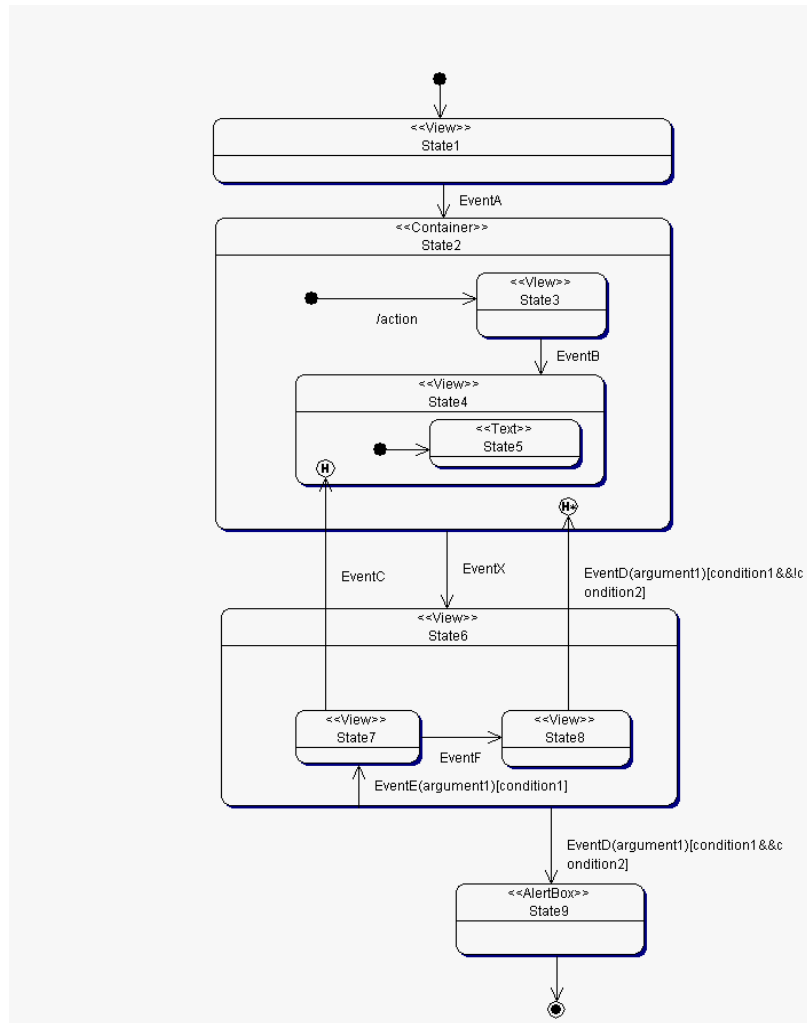


Figure 4 – Basic Elements of a Statechart

The transition annotation is written as Event(parameters)[guard condition]/action. All or any of these 4 elements can be present on a transition arrow. The small black dots are default start states. The circled dot is the end state.

Login Dialog Example

Figure 5 shows an example of a Statechart in use to describe the UI interaction, or navigation flow, for a simple login dialog which might appear at the beginning of any business application.

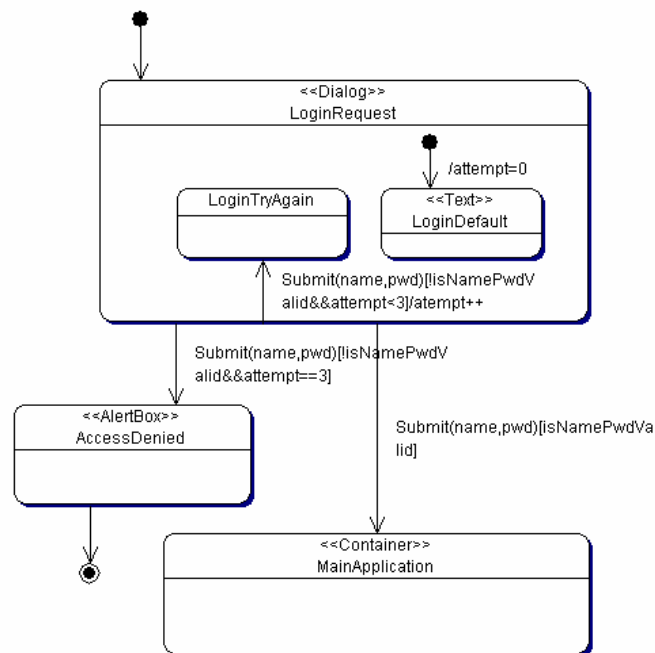


Figure 5 – Login Dialog Example

The Login Request dialog has two possible sub-states. The default state will prompt the user to login. The alternative will offer advice that the previous attempt was incorrect and request that the user try again. Hence, a single dialog box has two possible text messages. The text message to be displayed is modeled using the sub-states.

On initial entry to the dialog, the number of attempts is reset as an action of the default state transition to the initial sub-state – LoginDefault.

There is only one action possible from this login request – Submit. The action has three possible transitions. If the name and password are both correct then the application should flow to the MainApplication window. If there is an error in name or password but there are less than three attempts then the transition should lead to the LoginTryAgain sub-state of the LoginRequest dialog. The action on the transition will increment the

number of attempts counter. If there is an error in the name or password and the number of attempts is already 3 then the application will flow to a message informing the user that they have been denied access.

Exception handling

One aspect of interaction design that frequently gets less attention than it deserves is the interaction when the system fails. Particularly for web applications, the answer is usually just to create a single catch-all page saying “We are experiencing technical difficulties” or a similarly unhelpful, out of context response.

A simple extension to the notation, originally presented by David in his 2000 position paper for the TUPIS workshop at UML 2000 [Anderson 2000], allows exceptions to be modeled as another type of event that affect the state of the UI. The source of these events is not the client but the model. They are caught by an exception handling state which is a type of controller (or mediator). The exception handling state then transitions to a stable UI state based on an interpretation of what error occurred.

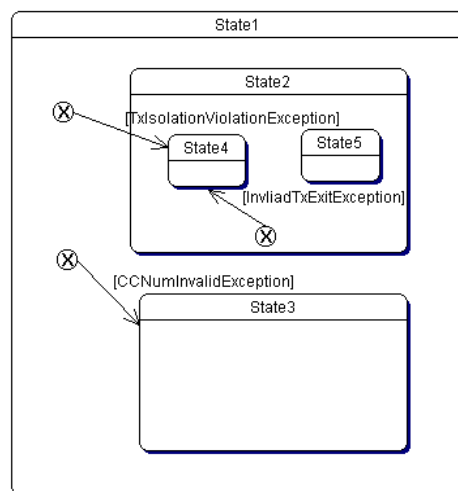


Figure 6 – Statechart with exception catch and handling transitions

The symbol used for exception states is an 'X' in a circle as shown in Figure 6. This was chosen to closely resemble the History State in the established Statechart notation. Transitions from the exception handler are labeled with the name of the exception they handle.

The introduction of the exception catching state provides a convenient shorthand notation for processing exceptions. Without this addition to the

notation, adding every possible exception to every possible event would greatly clutter the diagram and render it useless.

Patterns for Common Problems

Using Statecharts as a design approach is very amenable to the application of design patterns, due to its formality and rigor. The Login Dialog example in Figure 5 is a simple example.

Two more common problems in UI design which can be solved with patterns are presented below. Many others are possible. A pattern catalog of Statechart models can be developed for a project or organization. This facilitates re-use of design and code and delivers consistency in the user experience.


The 'Back button' Handler Pattern

Problem: When deploying a user interface as a web application there are some events and interactions available to the user outside the scope and control of the interaction designer. These are the navigation functions built-in to the web browser, such as the back, forward and bookmark features. It is common for web developers to disable these functions in transactional applications. This degrades the user experience and confuses users who are familiar with the browser functionality. Often messages such as “Do not use the ‘BACK’ button” are displayed prominently in the design. This has become accepted as a necessary constraint of web technology. However, the problem can be solved and Statecharts enable the foundation for a solution.

Solution: The JStateMachine framework must determine the state the user interface is in at the start of processing each incoming event. It can do this in one of two ways

- a. Examining the user session to determine the last state presented to the user.
- b. Examining the request and letting the client supply an identifier for the requesting state.

If the user activates one of the browser navigation features, these two sources will give different answers. The user session will contain the state the user should be in according to the interaction design. The client will supply the state the user believes she is in.



The application deployer can configure JStateMachine engine (described later) at runtime to use one or other of these sources, or to throw an exception if they differ. The exception result can be designed using the exception notation, also described later.

Consequences: There are three possible outcomes when the user uses the back button or other browser navigation feature, depending on the source JStateMachine uses for the source state.

1. Session: Undefined behavior. The framework will use the session-supplied state and search for the named event from that state. That event may or may not exist, and may not behave as expected if it does exist.
2. Client: The application will behave as the user most likely expects. The user will have navigated to a page and the framework will use the corresponding state. This is acceptable so long as a transaction boundary has not been breached.
3. Checked: The application will behave as designed by the interaction designer. The framework will throw an exception which will be handled through a designed exception handler.

The UI Transaction Pattern

Problem: There are scenarios where a user goal is reached through a series of interactions. These are often referred to as “conversational transactions.” A Wizard is a typical GUI pattern example for such a transaction. In Web design, lengthy forms are often divided over several pages to aid the user in focused concentration.

In such cases it is best that the user follows the steps in a consistent way, not starting in the middle or exiting unfinished. It is desirable to trap the case where the user leaves the transaction unexpectedly so that a warning dialog can be presented, or information given to date can be saved, or a transaction begun on an external system can be rolled back successfully.

Figure 7 demonstrates this with a banking application with a feature for applying for a mortgage online. It is desirable that the user enter and exit the mortgage application process at the appropriate points. If not then it is desirable that unexpected exit is trapped appropriately. A caught exception could display a screen recommending the user continue the transaction through a telephone or postal channel.

An exception handler can be defined to take whatever action we deem appropriate if the user enters or leaves this sequence unexpectedly.

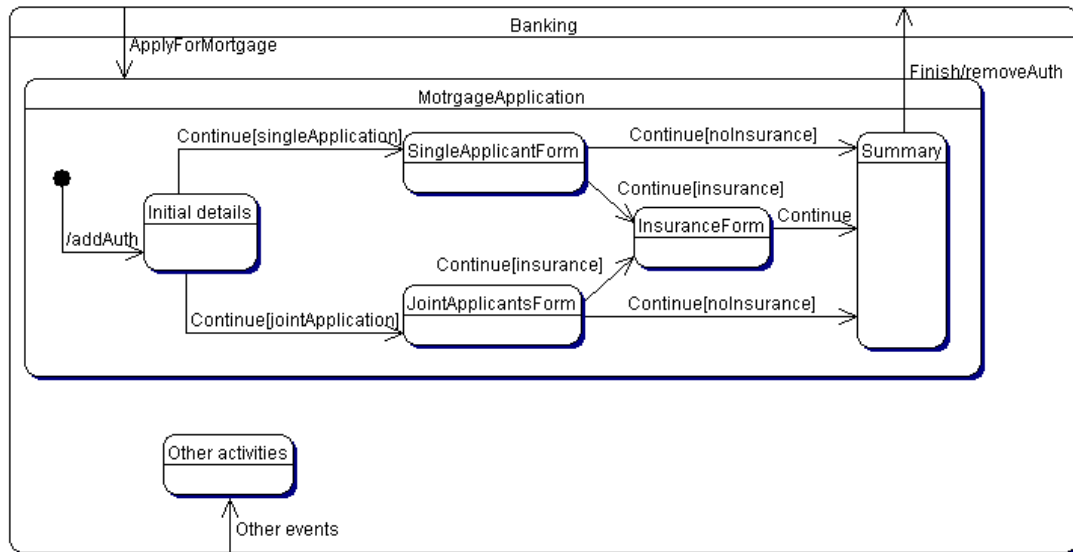


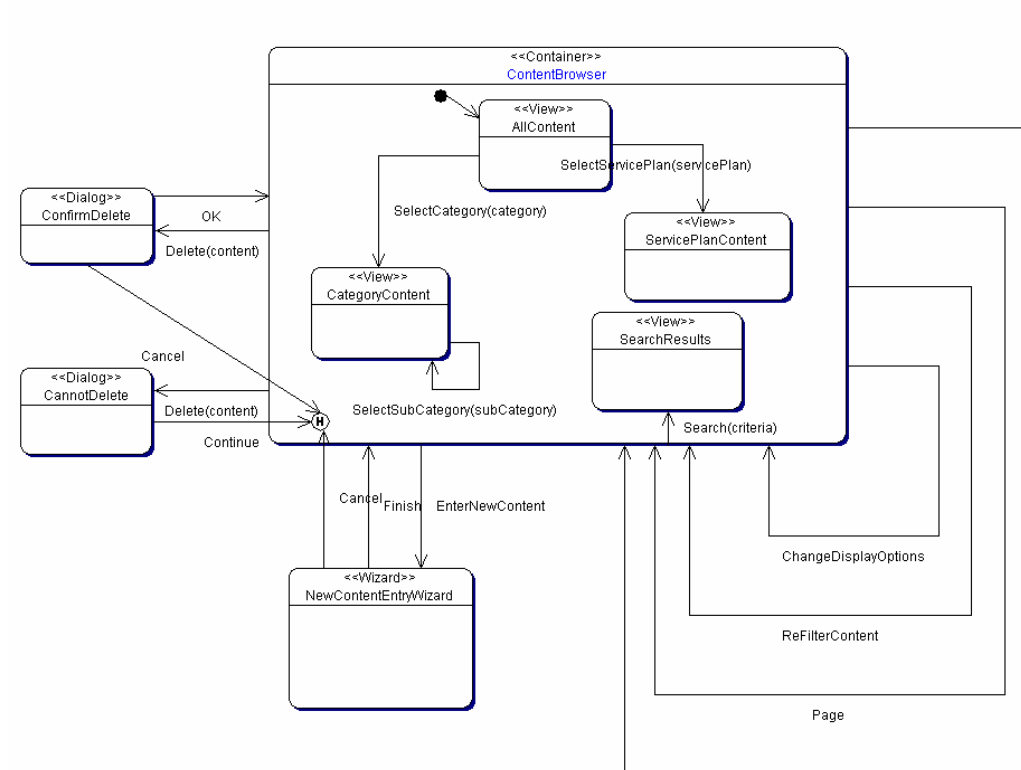
Figure 7 – Transaction state example

Solution: The access control mechanism in JStateMachine is designed to provide a nice solution to the UI Transaction problem.

Each state and event in a chart can be defined with required and forbidden access tokens. A user can enter a state or fire an event only if their session contains all required and no forbidden tokens for that state or event. If the access check fails the framework will throw an exception that can be handled as described above.

For each transaction in your interaction design define an access token. Every state and event within the transaction should have this as a required token, every other state and event should have it as a forbidden token. When the user enters the transaction through an expected transition, the side effects for that transition should include adding the access token to the user's session. When leaving the transaction in an expected way, the side effects should include removing the token.

Consequences: With this pattern if a user attempts to enter or leave a transaction through an unexpected route the framework will throw an access violation exception which can be handled as required.



The Statechart diagrams are translated directly into a Feature list of

Figures 9, 10 and 11 show the actual screen shots for the main views from Figure 8.

4thpassMAS

Home | Accounts | Services | **Content** | Devices | Settings

Content Browser | Categories | Reports

User, Role [Sign Out](#)

All Content
Browse content by category or service plan and manage content revisions. **1032 items listed**

Keywords: arcade game
Search

Browse By Category

- [Business & Economy](#)
B2B, Finance, Shopping...
- [Computers & Internet](#)
Internet, WWW, Software...
- [News & Media](#)
Newspapers, TV, Radio...
- [Arts & Entertainment](#)
Movies, Humor, Music...
- [Recreation & Sports](#)
Sports, Travel, Autos...
- [Health](#)
Diseases, Drugs, Fitness...

Browse By Service Plan

- [Basic Plan](#)
- [Family Plan](#)
- [Youth Plan](#)
- [Business Plan](#)
- [Premium Plan](#)

Display Options

Display entries: 25 per page

☒ Show summary
☐ Show details
☐ Show details and images

Apply

New Content Delete

< Previous | Page 2 of 7 | Next >

Content Type	Status	Device	Administrator
Screen Saver	Ready For Test	Motorola Accompli 008	Patrick Heddesten

<input type="checkbox"/>	Name	Content Provider	Downloads	Modified
<input type="checkbox"/>	U of Michigan Fight Song	Terra Vera	30	09-01-2002
<input type="checkbox"/>	John Madden 2003	Terra Vera	250	08-02-2002
<input checked="" type="checkbox"/>	Superball	Jupiter	28	08-03-2002
<input type="checkbox"/>	Death Race 2000	Terra Vera	15000	09-01-2002
<input type="checkbox"/>	Baseball Collage	Terra Vera	0	06-01-2002
<input type="checkbox"/>	Harvard Fight Song	Terra Vera	30	09-01-2002
<input type="checkbox"/>	Baseball USA	Terra Vera	250	08-02-2002
<input type="checkbox"/>	U of Michigan Fight Song	Jupiter	28	08-03-2002
<input type="checkbox"/>	John Madden 2003	Terra Vera	15000	09-01-2002
<input type="checkbox"/>	Superball	Terra Vera	0	06-01-2002
<input type="checkbox"/>	Death Race 2000	Terra Vera	30	09-01-2002
<input type="checkbox"/>	Baseball Collage	Terra Vera	250	08-02-2002
<input type="checkbox"/>	Harvard Fight Song	Jupiter	28	08-03-2002
<input type="checkbox"/>	Baseball USA	Terra Vera	15000	09-01-2002
<input type="checkbox"/>	U of Michigan Fight Song	Terra Vera	0	06-01-2002
<input type="checkbox"/>	John Madden 2003	Terra Vera	30	09-01-2002
<input type="checkbox"/>	Superball	Terra Vera	250	08-02-2002
<input type="checkbox"/>	Death Race 2000	Jupiter	28	08-03-2002
<input type="checkbox"/>	Baseball Collage	Terra Vera	15000	09-01-2002
<input type="checkbox"/>	Harvard Fight Song	Terra Vera	0	06-01-2002
<input type="checkbox"/>	Baseball USA	Terra Vera	30	09-01-2002
<input type="checkbox"/>	U of Michigan Fight Song	Terra Vera	250	08-02-2002
<input type="checkbox"/>	John Madden 2003	Jupiter	28	08-03-2002
<input type="checkbox"/>	Superball	Terra Vera	15000	09-01-2002
<input type="checkbox"/>	Death Race 2000	Terra Vera	0	06-01-2002

New Content Delete

< Previous | Page 2 of 7 | Next >

Copyright 2001-2002 4thpass, Inc. All rights reserved.

3.0.0 All Content

Figure 9 - MAS Content Browser – All Content View

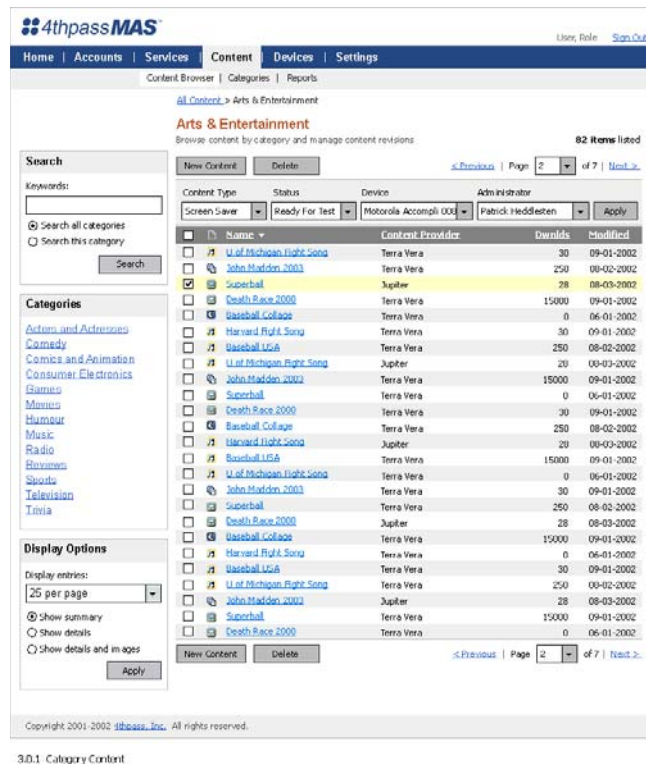


Figure 10 – MAS Content Browser – Category View

Design by Feature

The fourth stage of the FDD process consists of the development of a detailed UI specification including detailed designs such as those in Figures 9, 10 and 11 and additional description detailing the action and behavior in the event that the initial interaction model was insufficient.

Build by Feature

The final stage involves developing code which will actually build and display the required views and process the event actions as described in the model and detailed specification.

Part 2 of this paper (and presentation) will detail how this can best be achieved using the JStateMachine presentation framework.

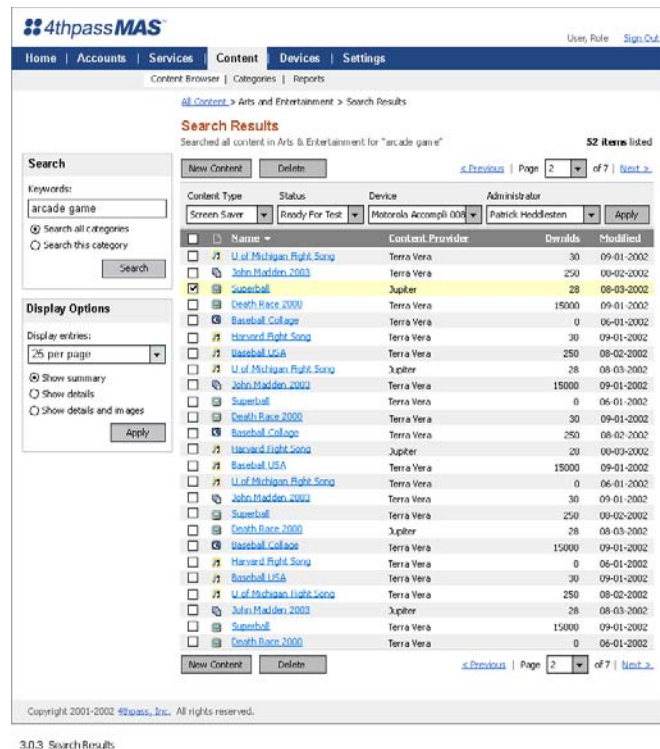


Figure 11 – MAS Content Browser – Search Results View

Part 2: Producing the implementation

Given that you have an interaction or behavioral design as described so far: How do you produce an implementation that definitely follows this design?

Background to the Type II MVC Pattern

The Model View Controller pattern is a widely-used approach to separating responsibilities for different aspects of an application into different pieces of code. The three components proposed by MVC are

- **Model:** Representing the business domain, the model encapsulates the business data model with the rules and actions that are defined in that domain.
- **View:** In displaying the user interface controls, the View fills a role that is conceptually very simple but often difficult to code.

- **Controller:** The Controller provides the behavior for the flow of the application user interface and mediates the interaction between the user and the model.

The Type II pattern is a variant of MVC that makes allowance for the one-way client-server structure of HTTP used by web applications. The interactions in type II MVC are

- **Model:** The model becomes a passive server, accepting instructions from the Controller and responding to queries from the Controller and View.
- **View:** The view may query the model for data, but takes its cue on the data and features to present to the user from the Controller.
- **Controller:** The controller handles all events that arrive from the user, asks the Model to perform actions as requested by the user, and advises the View on the result to display.

The division of responsibilities is best explained by the sequence diagram in Figure 12.

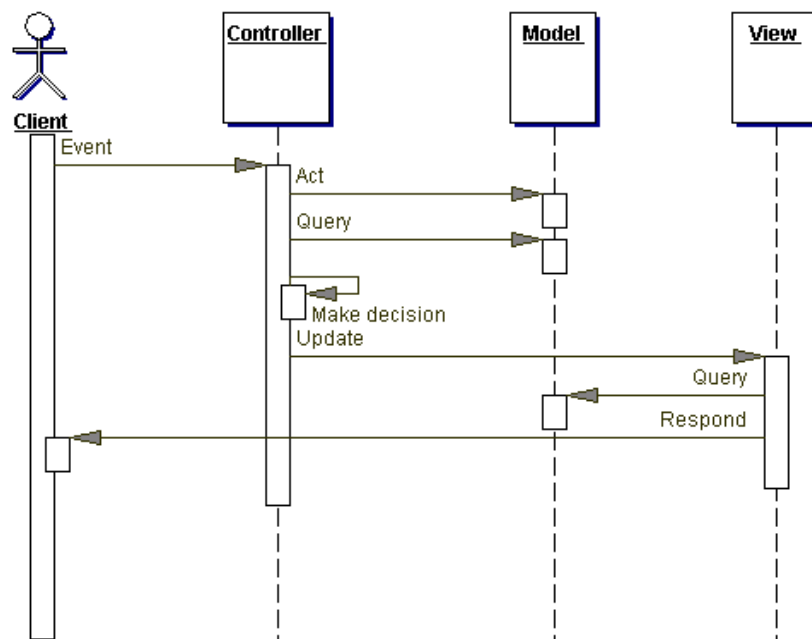


Figure 12 – MVC Sequence Diagram

The application framework

Consider an application framework that follows the Type II MVC pattern, refers to the interaction design at runtime, and delegates the details of the control and view logic to developer-supplied classes.

The design in this scenario is a UML Statechart that describes the user interface, with some information added. Events are associated with classes that act as handlers for the event, called the Controller classes. States are associated with classes that render the user interface for the state, called the View classes.

The sequence diagram in Figure 13 describes, in broad terms, how this framework works.

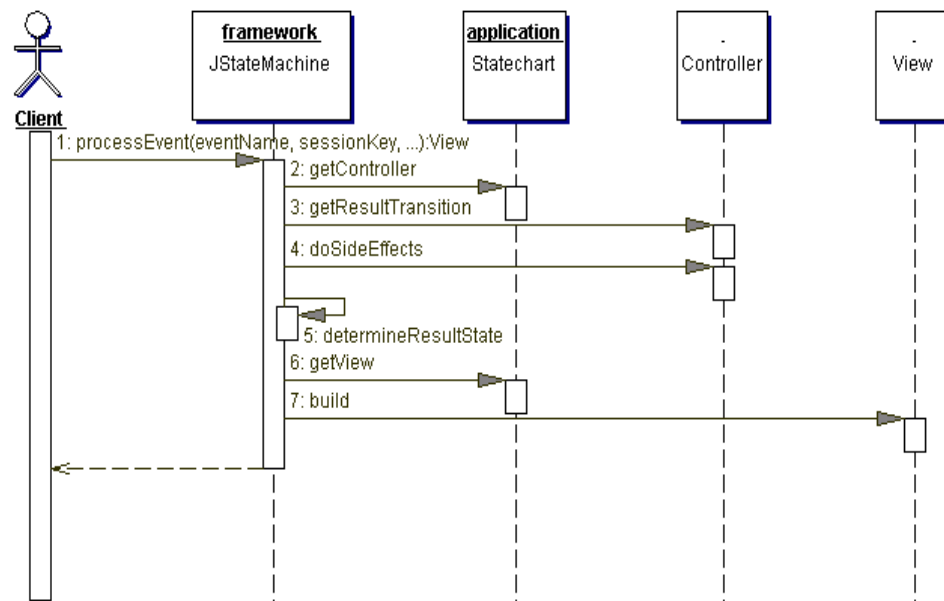




Figure 13 – JStateMachine framework sequence diagram

1. The event arrives from the client. This includes the name of the event, a session key to allow the framework to track the client's actions, and any parameters required by the event.
2. The framework uses the session to determine what state the client was in, and refers to the design (data encoded version of

- 
-
- the Statechart) to locate the event and load the correct controller class. In doing this, the framework will ensure that-
- a. The client is following a consistent path through the UI design and not jumping randomly from state to state. If an unexpected event arrives requesting a state inconsistent with the design then an exception will be thrown
 - b. The correct Controller class is loaded to handle the event. Any attempt to hack the client code and request a different controller will be caught and an exception will be thrown. This is a useful security feature.
 - c. c.(optionally) the client has the authorization to fire the event
3. The framework asks the Controller which transition to follow from the event. The Controller can interact with the model to execute the guard conditions and determine the transition flow.
 4. The framework gives the Controller a chance to perform any necessary actions (side-effects) associated with the transition.
 5. The framework follows the transition, handling history or deep (recursive) history pseudo-states, default events, or exception events to arrive at the correct next stable state. Optionally, it can determine whether the client (user) has permission to enter the state. If not an exception will be thrown. This allows role based authorization to be easily built into a JStateMachine implementation.
 6. The framework loads the correct View class for the resulting stable state.
 7. It then asks the View to build itself. The View may query the model at this stage to get data and information to be displayed to the client.
 8. The framework passes the built view back to the client as the result of the incoming event.

This structure provides an excellent division of responsibility both in the code and in the designer / developer roles.

In the code, the framework takes on all the common control tasks, including reading the design, loading the Controller and View classes, exception handling, access control (authorization), and session management.



Each Controller class will typically handle one event. It needs to choose the transition to follow from the event and perform any side effects. The framework will decide the result state, and the View has the single responsibility of rendering that state for the user.

Amongst the team roles, the interaction designer has the final say in the overall shape of the application. The developers can be divided cleanly between Model, View and Controller teams. Among the View and Controller teams Feature Sets can be based on small sections of the Statechart, giving very real and obvious service to the FDD assertion that Features should be small pieces of client-valued functionality.

JStateMachine Demonstration

The JStateMachine application framework has been built to follow this model, and will be demonstrated at ForUse2003. Aside from exception handling and access control, JStateMachine provides a feature that can help designers and business sponsors reach a common understanding of the design and its consequences.

The AutoViews are a set of View classes that read the chart given to JStateMachine and can produce a View of any state on the chart, including the events off the state with their parameters and the possible transitions. Any chart can be deployed using the AutoView to give users a prototype they can click through, giving a practical and hands-on understanding of the design.

A Friendlier Alternative to Statecharts

Statechart notation is very abstract. Although it has been used with success in several large companies, the notation may be too abstract for wider adoption. Some usability professionals may be more comfortable with alternatives developed amongst their own design and usability community, such as Jesse James Garrett's Visual Vocabulary (VV) notation. [Garrett 2000 & 2002]

Figure 14 shows the VV model for the Statechart in Figure 8. Visual Vocabularly is the interaction modeling tool of choice in step 1 of FDD at 4thpass Inc.

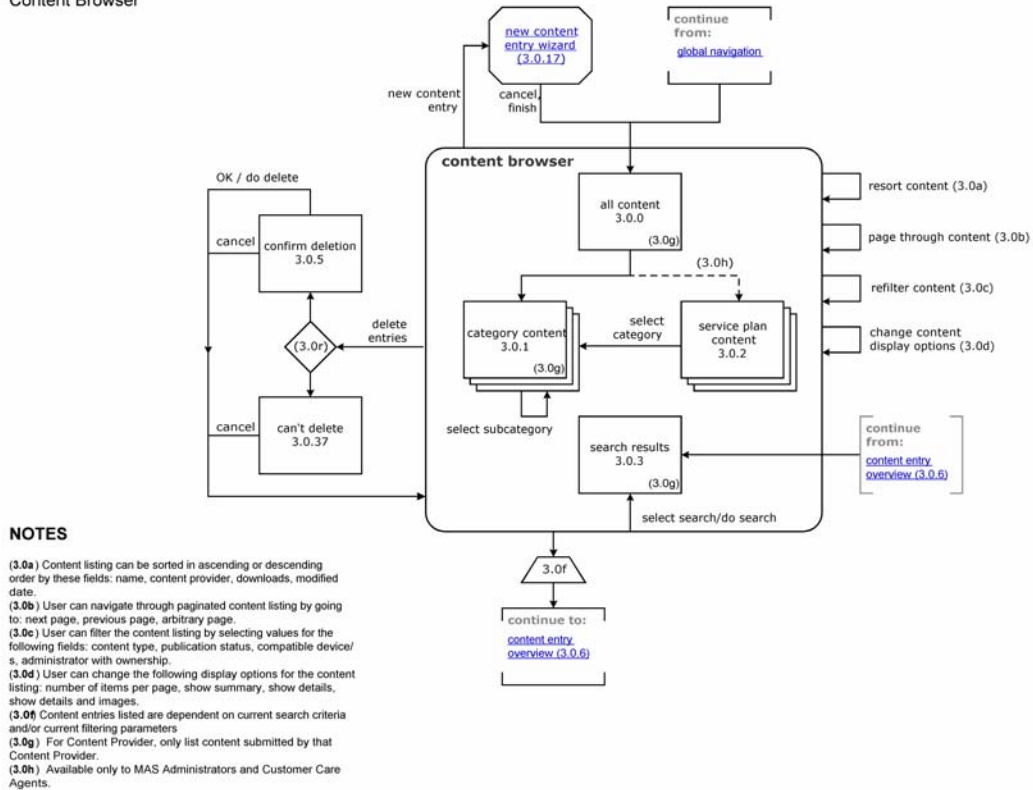


Figure 14 – Visual Vocabulary Diagram for MAS Content Browser

Acknowledgements

Figure 1 is provided by the kind courtesy of Stephen R. Palmer and is used with permission. All rights reserved. Copyright Step-10 Limited, <http://www.step-10.com/>
Figure 9, 10, 11, and 14 were created by Randy Moss of 4thpass as part of his work as lead product designer for Mobile Application Server (MAS). All rights reserved. Copyright 4thpass Inc., <http://www.4thpass.com/> .
JStateMachine can be found on the web at <http://www.jstatemachine.org/>

References

- Anderson, David J. (1999], *Server-side MVC Presentation Layer*, uidesign.net
Anderson, David J. (2000), *Extending UML for UI*, uidesign.net
Coad, Peter, Le Febvre, Eric and De Luca, Jeff (1999), *Java Modeling in Color with UML – Enterprise Components and Process*, Prentice Hall
Constantine, Larry L. (2001), *Process Agility and Software Usability: Toward Lightweight Usage Centered Design*, Proceedings of OOPSLA
Garrett, Jesse James (2000), *A Visual Vocabulary for Describing Information Architecture and Information Design*, <http://www.iig.net/ia/visvocab/>
Garrett, Jesse James (2002), *The Elements of User Experience: User Centered Design for the Web*, New Riders
Goldratt, Eliyahu M. (1997), *Critical Chain*, The North River Press
Harel, David (1987), *Statecharts: a visual formalism for complex systems*, Science of Computer Programming, 8(3), 231-74
Horrocks, Ian (1999), *Constructing the User Interface with Statecharts*, Addison-Wesley Longman
Leach, Lawrence P. (2000), *Critical Chain Project Management*, Artech House
O’Byrne, Brían (2003) *JStateMachine* was first implemented in 1999 for Ebeon.com a subsidiary of Eircom, a traditional telecommunications company in the Republic of Ireland, for Eircell, a wireless telecommunications company, now known as Vodafone Ireland.
Reinertsen, Donald G. (1997), *Managing the Design Factory: A Product Developer’s Toolkit*, Free Press