

MetaEdit+ Watch Example チュートリアル

概要

このチュートリアルは、MetaEdit+を使って作成されたモデリング言語の内容を理解する事で、モデリング言語の構築に関する考え方とその手法を学習することを目的に書かれています。そのため、モデリング言語を構築する為の具体的な手順（MetaEdit+の各機能の使い方）には詳しく触れてはいません。この手順に関しては、別途資料が用意されていますのでご請求ください。ご請求に関する情報は資料の末尾に記載されています。

1 Watch Example について

Watch Example は、デジタル腕時計のソフトウェアアプリケーションの実装を目的としたモデリング言語とそのツールです。これは MetaEdit+に対してドメイン毎のモデリング環境（以下、DSM 環境）を組み込む方法の一例になっています。ここでは組み込みやリアルタイムシステムの為に設計された言語について紹介していますが、この例の基本原理は他のドメインにもうまく適用することができます。

この章では、Watch Example とその実装の背景になる考え方を紹介します。その後、2章で Example の操作方法と Watch モデリング言語の使い方を説明します。3章では、Watch Example のアーキテクチャーの詳細な解説を行います。

Watch Example を始めるには、MetaEdit+の使用法に対する基本的な知識が必要です。そのためにはマニュアル内の Evaluation Tutorial - Family Tree Example が最適です。

1.1 Watch Example の基本的な考え方

時計のモデリングのために、既存の汎用モデリング言語や標準のモデリングツールを使うのではなく、DSM 環境を構築する理由とは？ 標準技術を適用することが可能であるにも関わらず、DSM を使用することでより多くの利益が得られます。以下、ソフトウェア開発で DSM を使用することで得られた成果の報告を検証してみましょう。

- 1) **生産性が10倍程度向上します。**従来のソフトウェア開発では、その過程で何度かのマッピングが必要となり、これがエラーを引き起こす要因になっていました。最初にドメインのコンセプトがデザインにマップされ、その後さらに、プログラミング言語にマップされます。つまり、同じ問題を繰り返し解決することになります。DSM はドメインコンセプトを最適な抽象化レベルのモデルにマッピングします。このモデルから最終的な製品が自動的に生成される為、その後のマッピングが不要になります。製品開発を行う為の学習時間も、従来の開発手法の学習時間に比べて、5 ~ 10倍早くなります。
- 2) **高い柔軟性と変更への素早い対応が得られます。**ソースコードよりも設計に重点を置く為に、仕様変更にも素早く対応できるようになります。ドメインコンセプトのレベルなら変更は非常に簡単になり、その変更を反映する単一のモデルからあらゆるプラットフォームに対応したソースコードや、製品のバリエーションに対するソースコードを生成できます。
- 3) **ドメインの専門知識を、開発チーム全体で共有できる仕組み。**チーム開発を行う場合、開発者のドメインに対

する知識不足が問題になります。新しい開発者が、製品開発を行う為に十分な知識を得るには長い学習時間が
必要です。より経験を積んだ開発者でさえ、ドメインのエキスパートによる助言が度々必要になるでしょう。
このチュートリアルで紹介しているアプローチでは、エキスパートがドメインのコンセプト、ルール及びソー
スコードへのマッピングを定義します。開発者は定義されたルールとコンセプトに従ってモデルを作り、コー
ドは自動的に生成されます。

開発に以下に示す条件が求められる場合に、これらが非常に重要になります。

- ドメイン固有の知識が必要
- 製品ファミリー（似通った製品バリエーション）がある
- 中規模・大規模な開発チーム
- 短期間での製品化が要求される
- 非常に高い品質が求められる

独自のドメインでソフトウェアを開発する為の DSM 環境を構築することで、これらの利益をうまく享受できる
方法を、Watch Example を使って紹介します。以降で 2 段階に分けて紹介を進めます。最初に環境を構築する方
法を説明します。次に、特定ドメインのアプリケーション開発に使用する方法を紹介します。

1.2 Watch Example の構築

Watch Example 例は、あらゆる DSM 環境の構築に応用することができます。環境を構築するときには、課題と
なるドメイン、ターゲットプラットフォーム及び、それらの橋渡しとなる DSM 環境の 3 つが常に存在します。ド
メインとターゲットプラットフォームは既存の部品である為に、それらの鍵になる実体を探し出しマッピングを定
義することで DSM 環境を構築します。

この例では、他のドメインの開発者が簡単に理解できるような一般的に十分に知られているドメインが必要でし
た。そこで、幾つかの製品ラインナップを持ったデジタル腕時計のドメインを選択しました。そして、モデリング
によって腕時計を開発し、そのモデルから完全に動作するプログラムを生成する DSM 環境を構築しました。

このドメインは非常にシンプルなので、ドメインコンセプトの鍵が非常に簡単に見つかりました。デジタル腕時
計は、物理的な Display ユニットと振舞を規定する Logical Watch に分けられます。Display ユニットは腕時計の
ボタンと表示装置の構成を定義します。表示装置は（アラームのような）サービスが提供されているか否かを示す
アイコンと、時・分・秒の様な基本的な時間単位を表示する 2 桁の表示ゾーンのどちらか又は両方を持ちます。
Logical Watch アプリケーションは、アラームやストップウォッチといったサブアプリケーションの組み合わせで、
必要な機能を定義します。

この方法で腕時計を分解すると、非常に良い再利用性が得られます。Logical Watch と Display のサブアプリケ
ーションは各々独立して定義され、事前に定義されたインターフェースを介して通信します。これらはコンポー
ネント部品です。サブアプリケーションを組み合わせる事で新しい Logical Watch を作り、それを（既存又は新規の）
Display と組み合わせる事で、開発者は新しい腕時計のバリエーションを素早く構築できます。この腕時計構成の
考え方を図 1 - 1 に示します。

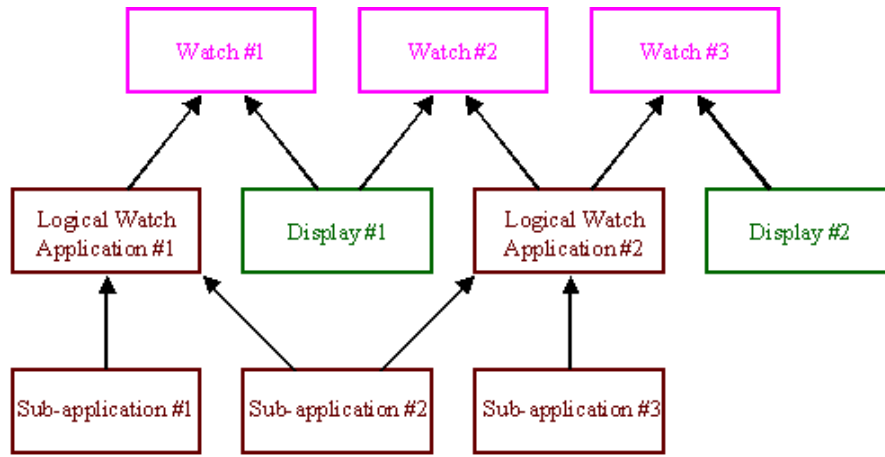


図 1-1. A watch configuration

現実にはプラットフォームは MPU に制御される電子デバイスになるでしょう。実行環境を用意できない場合も十分考えられます。そのために、生成されたコードの実行と腕時計の機能テストを行う為に、最終製品と同じように動作するテスト環境を構築しました。この手のソフトウェア開発では、このようなテスト環境がよく利用されます。

Watch Application のモデリング言語はステートマシンを基本にしています。これは組み込みソフトウェアをモデル化する従来どおりの方法です。ただしステートマシンのセマンティクスを大幅に拡張して、より高い表現力を得るようにしました。この拡張によって、有限ステートマシンでアプリケーションロジック全体を記述でき、100%実行可能なソースコードが生成可能です。

Watch モデリング言語は2種類のダイアグラムタイプで構成されています。先ず腕時計ファミリーの中の製品ラインナップを記述する Watch Family があります。このグラフタイプでは、モデルを作成するために使用する Logical Watch アプリケーションと Display も記述します。Logical Watch アプリケーションとそのサブアプリケーションは WatchApplication ダイアグラムタイプを使って記述され、ここでステートマシンを使ってアプリケーションを実装します。WatchApplication ダイアグラムは、ドメイン固有のコンセプト（ボタンやアラーム等）を利用できるようにセマンティクスをカスタマイズし、ドメイン固有の追加を施した特殊な状態遷移図です。WatchApplication ダイアグラムの例を図1-2に示します。

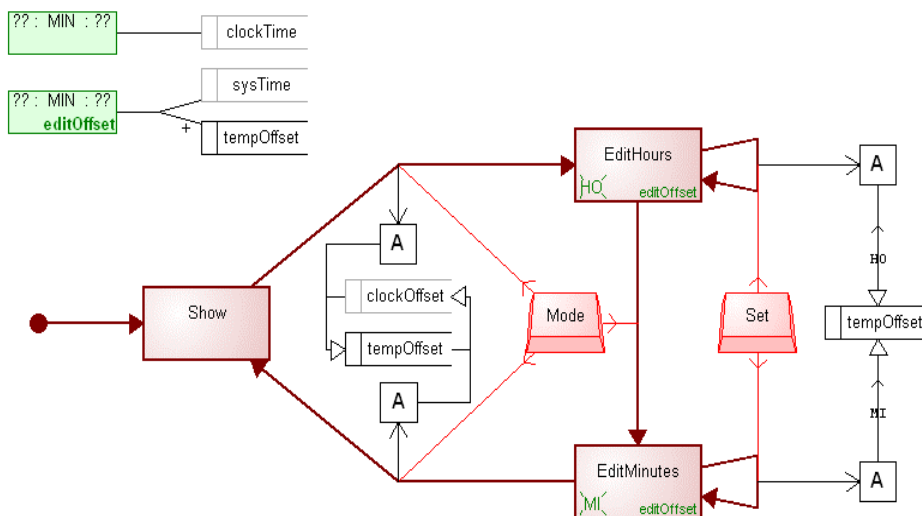


図 1-2. An example of a WatchApplication diagram.

あらゆるプラットフォームで実行できるテスト環境が欲しかった為、実装言語として JAVA を選択しました。JAVA クラスのドメインフレームワークを、標準の JAVA ランタイムに追加する形で作成しました。このフレーム

ワークは、全ての腕時計に共通して利用され、ユーザーインターフェースと Display とステートマシンの Abstract スーパークラスを提供します。そして、これらのコンポーネントを有効に利用できるように、コードジェネレータを実装しました。例えば、これらのサブクラスを作ることなどで。

最初の実用的な Watch モデリング言語(と1つの完全な腕時計のモデル)の設計と実装には、2人の開発者によるチームで8人日が必要でした。この開発者は、JAVA プログラミングの経験も、腕時計のソフトウェアの開発経験もなく、また社内には既存の腕時計用ソフトウェアコンポーネントもありませんでした。JAVA フレームワークの開発に5人日、モデリング言語開発に2人日、コードジェネレータの設定に1人日必要でした。これらの時間には、設計、実装、テスト及び、基本ドキュメントの作成が含まれています。環境が構築された後、新たな腕時計の製品を15分で作れるようになりました。手作業で最初の腕時計を開発するには5~6人日が必要と推測され、各追加製品の開発に1人日が必要と考えられます。従って、3種類目の腕時計を完成した時点で、開発コストが回収できると考えられます。

2 Watch Example を使う

この章では、Watch Example へのアクセス方法と使用方法(最初に既存のモデルの扱い、次に独自のモデルと機能の作成、最後にモデリング言語の拡張)を説明します。

2.1 Watch Example にアクセスする

Watch Example にアクセスするには、MetaEdit+を実行し、demo レポジトリと'watch'プロジェクトを選択してログインします。ログイン後、グラフブラウザーやダイアグラムエディタといった MetaEdit+のツールを使って Watch Example にアクセスできます。

2.2 Watch Example を使ってみる

MetaEdit+メインランチャーのグラフブラウザー画面を見てください。Watch Example に関連した全てのモデルが Graphs リストボックスに表示されています(Projects リストボックスの Watch が選択されていることを確認してください)。2007Models:WatchFamily ダイアグラムを選択し、ダブルクリックするか又は、右クリックで表示されるポップアップメニューから Open を選択することで、ダイアグラムが開きます(図2-1参照)。

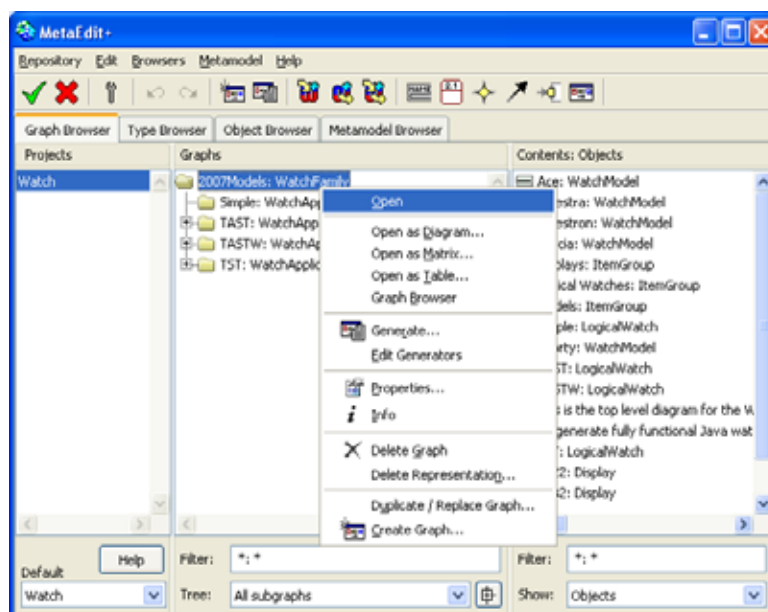


図 2-1. Opening a WatchFamily diagram.

このダイアグラムは腕時計の製品ラインナップを表しています。実際の製品ラインナップが上部の Models グループに表示され、LogicalWatch のアプリケーションと Display コンポーネントが下部に表示されています。モデル要素のプロパティにアクセスするには、各要素をダブルクリックするか、要素を選択した上で、ポップアップメニューから Properties.. を選択します。

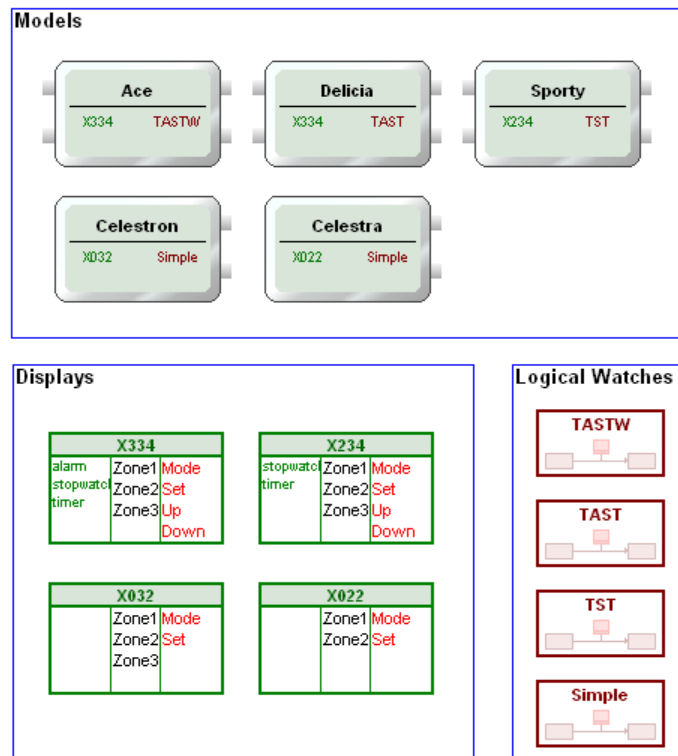


図 2-2. The '2007Models' WatchFamily diagram.

Watch Example の詳細を説明するために、LogicalWatch のアプリケーションの構造を見てみます。Logical Watches グループから 'TASTW' を選択し、ポップアップメニューから Decomposition... を選択します。その後表示されるダイアログで Open ボタンをクリックします。'TASTW':WatchApplication ダイアグラムが表示されます(図 2 - 3 参照)。グラフブラウザからダイアグラムを開くことも出来ます。

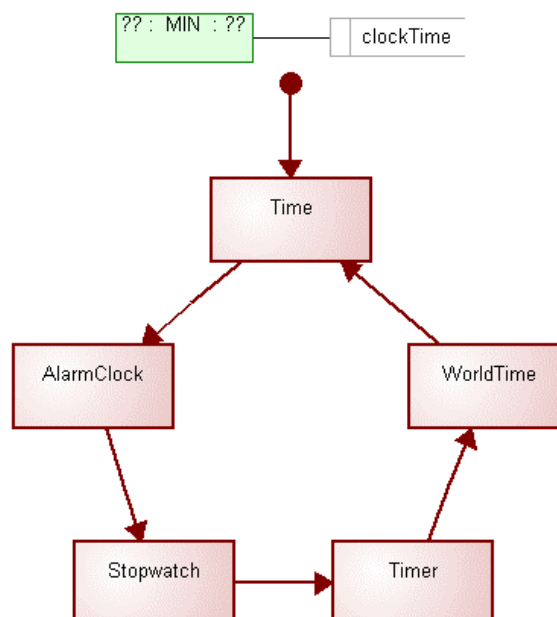


図 2-3. The 'TASTW' WatchApplication diagram.

このダイアグラムは LogicalWatch のアプリケーションの構造を表しています。ここで表現されているのは、サブアプリケーションのトップレベルの論理的な構造のみで、基本的に、その LogicalWatch が包括しているサブアプリケーションの種類と呼び出される順番を規定しています。LogicalWatch を開始した時（例えば、時計の電源を入れた時）に、最初の 'Time' サブアプリケーションが呼び出されます。'Time' サブアプリケーションを抜けると、'AlarmClock' サブアプリケーションが呼び出されます。このサイクルは、'WorldTime' サブアプリケーションを抜け出し、'Time' サブアプリケーションが再び呼び出されることで完結します。'TASTW' の名前は、これらのサブアプリケーションの頭文字を順に繋いで出来ています。各サブアプリケーションの詳しい定義は、LogicalWatch を選択して Decomposition することで参照できます（サブアプリケーションのモデルダイアグラムが開きます）。

これで、テスト環境で実行可能な腕時計のソースコードを生成する準備が出来ました。全ての WatchApplication ダイアグラムを閉じて、WatchFamily ダイアグラムに戻ります。コード生成を行う前に、生成すべきコードのプラットフォームを選択しなければなりません。Graph-> Properties... を選択してプロパティダイアログを表示します。アプリケーションを実行する OS を選択（Windows 又は Linux）します。正しいバージョンの J2SE がインストールされていることも確認してください。

コード生成を実行するには、Graph->Generate...メニューを選択、或いは、ツールバーの Generate ボタンをクリックし、表示されるリストから 'Autobuild' を選択します。このジェネレータは頻繁に呼び出される為、Build ボタンがツールバーに配置されています。Autobuild を実行すると、全ての腕時計の JAVA コードが生成され、コンパイルされ、ウェブブラウザ上のテスト環境で実行されます（図 2 - 4 参照）。

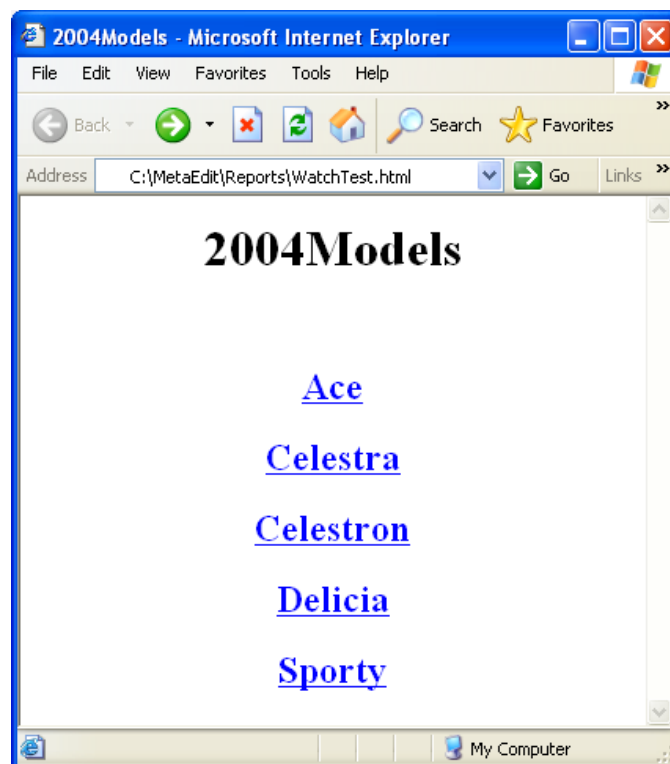


図 2-4. The watch test environment.

ウェブページから腕時計のラインナップを選択し、そのテスト環境を開く事で各腕時計のテストが可能です。ウェブブラウザのセキュリティー設定に依存しますが、選択した腕時計が実行されるまで、幾つかのダイアログやクエリーを通過しなければなりません。最終的に図 2 - 5 のように腕時計が実行されます。腕時計が動作しない場合は、JAVA の実行環境を確認してください。

腕時計のテスト環境でボタンを押してアイコンや時間の表示上の振舞を確認することで、腕時計のテストを行います。アプリケーションの名前はグレー領域の上部に表示され、ページ下部に現在のステータが表示されます。

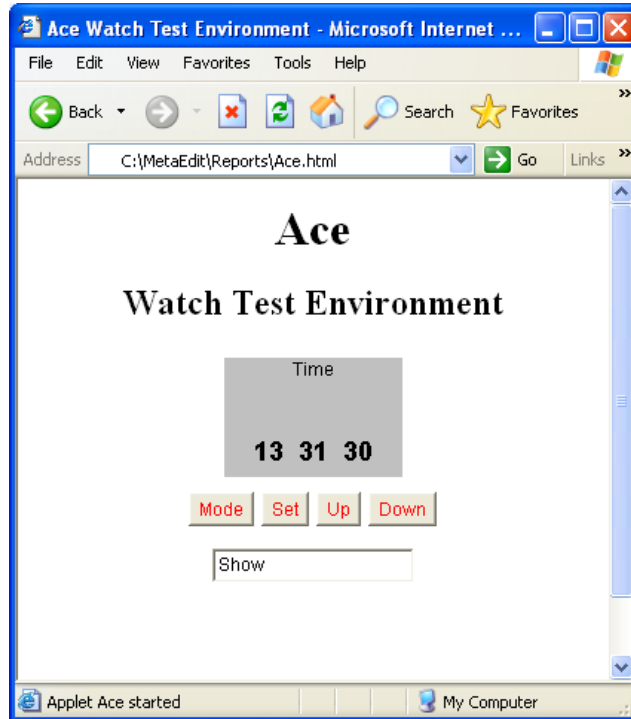


図 2-5. The test environment for a watch model.

幾つかの腕時計ラインナップのみを選択してコードを生成することも可能です。WatchFamily のグラフプロパティを開き、Selected Models リストボックスに腕時計を入力する事でコードを生成する腕時計を選択可能です。腕時計を入力するには、ポップアップメニューから Add Existing...を選択します。その後開くダイアログで必要な腕時計ラインナップが選択できます。'Autobuild'を実行すれば、選択した腕時計のコードのみが生成されます。

JAVA コードに加えて、技術ドキュメントをモデルから直接生成することも可能です。ジェネレータ実行時に'Watch family Documentation'を選択することで、ドキュメントを生成できます。生成されたドキュメントはウェブブラウザを使って表示可能で、ドキュメントにはハイパーリンクが設定された図とテキストがあります。

2.3 新しい腕時計ラインナップを作成する

次に、Watch Example を使って、新しい腕時計を開発します。既存の表示と LogicalWatch アプリケーションを単純に結び付けることが、最も簡単な方法です。しかしながら、ここで少しだけ面白い試みを行いましょう(4つの表示領域と2つのボタンを持ったストップウォッチだけの腕時計を作ります)。現状では、4つの表示領域と2つのボタンを持った Display は存在しないため、新たに作成する必要があります。

まず WatchFamily ダイアグラムを開き、新たな Display オブジェクトを作成します。Display の名前(ここでは X042 を使用)を入力し、UnitZones リストボックスに4つの Zone (Zone1~3はポップアップメニューから AddExisting...を選択して既存のものを再利用可能、Zone4は AddElement..で新規に作成が必要)と2つのボタン(既存のものから'Up'ボタンと'Down'ボタンを再利用)を追加します。プロパティダイアログは図 2 - 6 のようになります。OK ボタンでダイアログを閉じます。

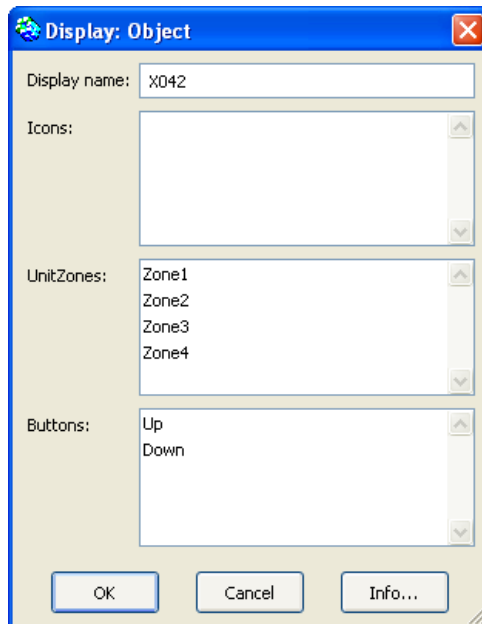


図 2-6. Property dialog for the new Display object.

次に必要なコンポーネントは、ストップウォッチのための LogicalWatch アプリケーションです。Stopwatch サブアプリケーションが再利用可能ですが、それを LogicalWatch アプリケーションとして利用する前にパッケージングする必要があります。WatchFamily ダイアグラムで、新たな LogicalWatch オブジェクトを作成し、プロパティに 'StopWatch' WatchApplication のグラフを結び付けます。作成した LogicalWatch を選択し、ポップアップメニューから Decompositions... を選択します。表示されるリストから 'StopWatch' を選択します。これで、LogicalWatch アプリケーションが定義できました。ここでは、トップレベルの WatchApplication の状態図を介さずに、LogicalWatch から Stopwatch サブアプリケーションを直接使用しています。

新しい腕時計を完成させるには、新たな Display と Logical Watch を WatchModel に結合する必要があります。WatchFamily ダイアグラムで、新たな WatchModel オブジェクトを作成します。名前（ここでは 'JustStopwatch' を使用）を入力し、新しく作った Display と LogicalWatch をプロパティに結び付けます。ダイアグラムは図 2 - 7 のようになります。

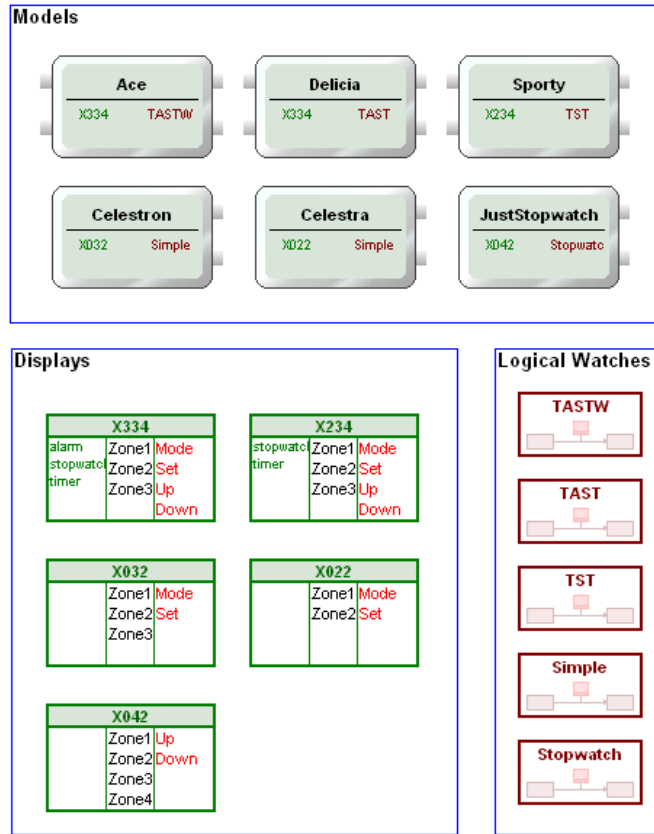


図 2-7. The WatchFamily diagram with new objects.

2.4 Watch Model に機能を追加する

最後に、ステートマシンの定義を修正することで、既存のサブアプリケーションに機能を追加します。Graph Browser で、Stopwatch:Watch Application を開きます。ダイアグラムは図 2-8 のように表示されます。

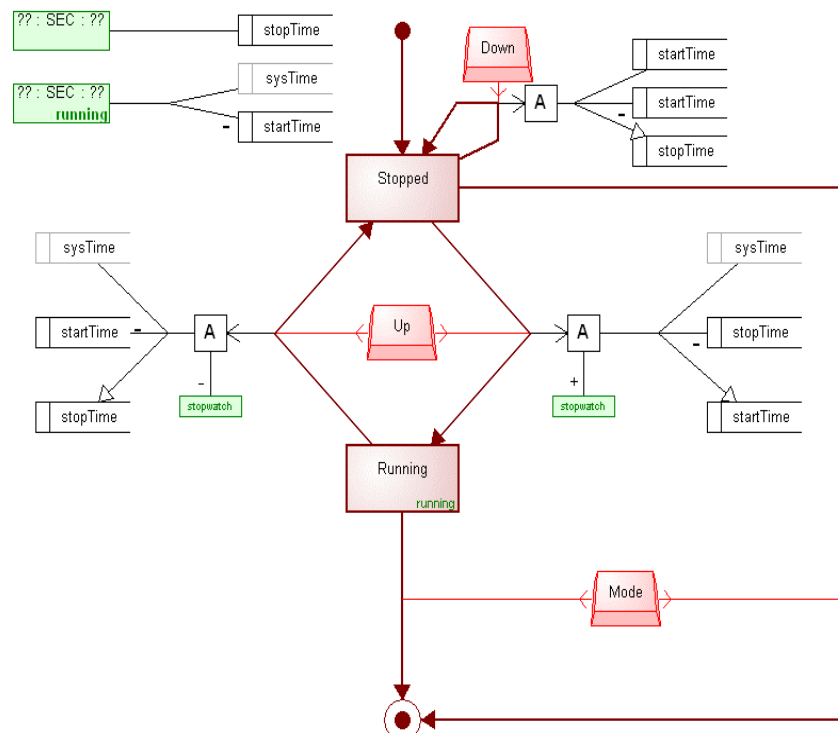


図 2-8. The 'Stopwatch' WatchApplication diagram.

'Stopwatch'という名前のサブアプリケーションは、様々なイベントの時間の長さを計る（いわゆるストップウォッチ）為のアプリケーションです。このアプリケーションがアクティブになると、'Stopped'のステートが表示され、ゼロの時間カウンターが表示されます。ここからの経路は3つあります。'Mode'ボタンを押すと、アプリケーションがアクティブでなくなり、トップレベルのステートマシーンに戻ります。'Down'ボタンを押すとカウンターの値がリセットされます。'Up'ボタンを押すと Starttime を設定して'Running'ステートに移行することで、カウンターがスタートします。再度、'Up'ボタンを押すと、 Stoptime を計算して'Stopped'ステートに移行することで、カウンターが停止します。'Running'ステートの間に'Mode'ボタンを押すことでもアプリケーションを終了できます。

しかしながら、このストップウォッチサブアプリケーションには、幾つかの不足があります（ラップタイム機能がありません）。このようなラップタイム機能を追加する為に、'Running'ステートがアクティブの時に'Down'ボタンが押されれば、ラップタイムを計算した後'LapTime'ステートをアクティブにし、ラップタイムを表示するといった処理を追加します。再度'Down'ボタンが押された場合、制御が'Running'ステートに戻されるようにもします。

新しい機能に追加する前に、Display に時間を表示する方法の基本的な仕組みを説明します。各々のアプリケーションステートは、アクティブ時に表示する時間の計算方法を規定した表示機能を参照しています。この表示機能は DisplayFn オブジェクト（図2 - 8、左上にある2つの緑の四角形）で表されており、各々、複数のステートで共有することが可能です。ステートのシンボルの右下にある緑のテキストは、どの表示機能を参照しているかを表しています。ストップウォッチアプリケーションには、2つの表示機能があります。ひとつは'Running'で、もう1つは名前がありません。名前を付けない場合はデフォルトの表示機能を意味します。表示機能の名前が明確でない全てのステートは、名前のない表示機能を参照します。

表示機能は、Display の中心に表示される時間単位もセットします。この機能により、3つの表示場所（Zone）に、標準の時計アプリケーションでは時・分・秒を表示し、ストップウォッチでは、分・秒・100分の1秒を表示することが可能です。

JAVA プラットフォームに組み込まれたサービスである変数と参照変数を基本にして、実際の時間を計算しています。例えば、'Running'は、システム時間を保持している参照変数'sysTime'から変数'startTime'を引き算して求められる時間を表示します。

ラップタイム機能を構築するための最初の作業は、新しいステートオブジェクトを作成し、'LapTime'（日本語でラップタイムとすることもできます）という名前を付けることです。ここでは、デフォルト表示機能が使用可能で表示を点滅させる必要もないため、名前以外のプロパティは空白のままにします。次に、'Running'ステートから'LapTime'ステートへの Transition と、'LapTime'ステートから'Running'ステートへの Transition を作成します。

続いて、これら両方の Transition に対するトリガーイベントとして、'Down'ボタンを関連付けます。新たにボタンを定義する必要はなく、既存のボタンを再利用できます。既存の'Down'ボタンを選択し、CTRL+C キーでコピーし、CTRL+V キーで貼り付けます。（このボタンをダブルクリックして名前を“ダウン”に変えてみるのもアイデアです。その場合、元のボタンも反映されることでしょう） 貼り付けられたボタンは、カーソルと一緒に動きます。'LapTime'ステートの近くにカーソルを移動し、クリックすることでボタンを配置します。このボタンを'LapTime'ステートと'Running'ステートの間にある両方の Transition に関連付けます。各々のリレーションシップを選択し、右クリックで表示されるポップアップメニューから'Add a New Role...'を選択します。そのまま'Down'ボタンをクリックしてロールを接続します。ダイアグラムは以下のようになるでしょう。

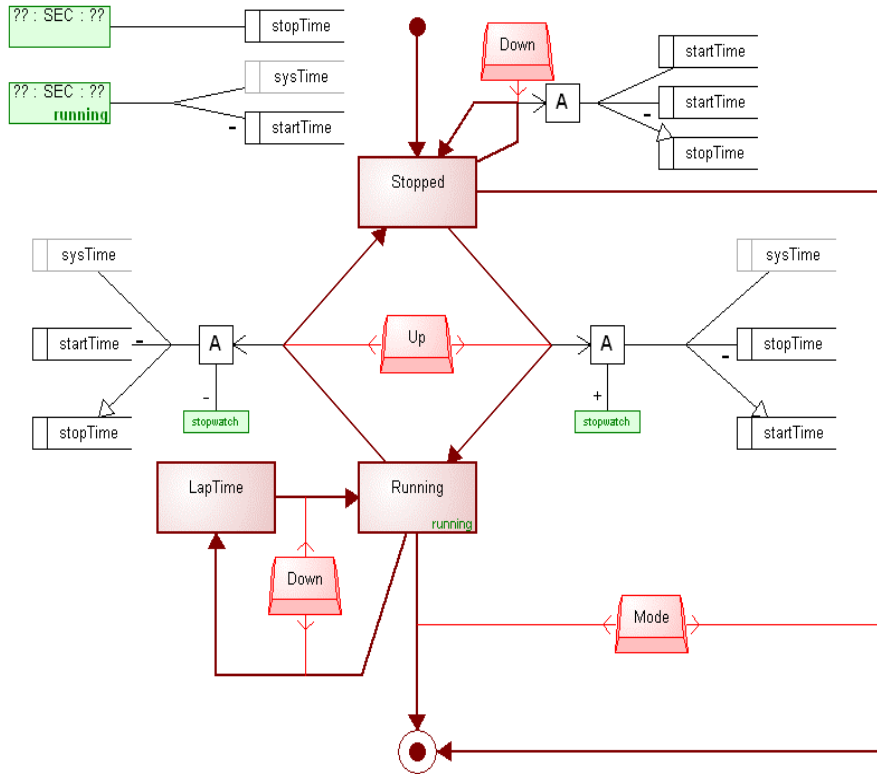


図 2-9. The 'Stopwatch' diagram with definitions for a new state.

最後に、'Running'ステートから'LapTime'ステートに移るときにラップタイムを計算する Action を定義しなければなりません。最初に、Action オブジェクトを作成し、'Running'ステートから'LapTime'ステートに遷移する Transition リレーションシップからロールを接続します。次に、参照変数'sysTime'と変数'startTime'と'stopTime'を再利用して、以下のダイアグラムと同じように接続します。

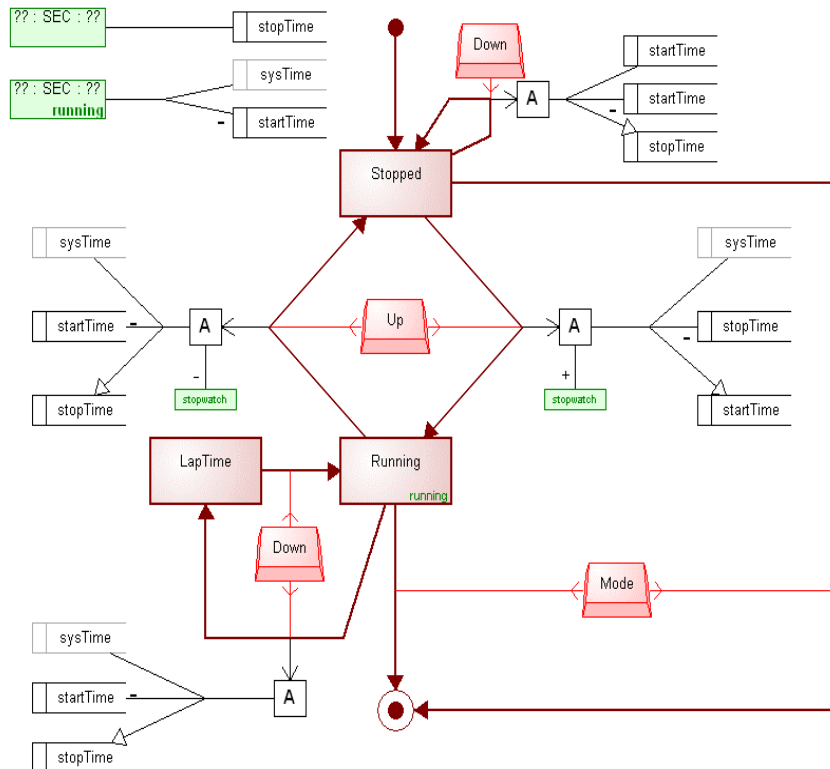


図 2-10. The extended version of 'Stopwatch' sub-application.

Action、Variableref 及び Variable の各オブジェクトの Calculation リレーションシップを定義するには、シフトキーを押しながら、'Action' 'sysTime' 'startTime' 'stopTime' の順にオブジェクトをクリックし、右クリックで表示されるポップアップメニューから Connect を選択します。選択可能なリレーションシップの組み合わせを表示するダイアログが出るので、'Set (ActionBody Action) (Get sysTime) (Minus startTime) (Set stopTime)' を選択します。次に出てくるダイアログは変更せずに OK します。これで、ラップタイム機能の定義は完了しました。追加した計算によって 'stopTime' に保存された値は、'LapTime' ステートに結びついたデフォルト表示機能が参照しています。これがラップタイムを正しく表示する仕組みです。コードを生成してテスト環境で実行してみましょう。

2.5 Watch モデリング言語の拡張

作成したモデリング言語はその後どうなるでしょう？この様なモデリング言語は、ドメインやプラットフォームの変化に対応し続け、モデリング言語自身も進化し続けなければなりません。従って、将来起こりうる要求に対応できるような DSM を開発できることが重要です。Watch モデリング言語を開発した方法を使えば、(モデリング言語の定義、コードジェネレータ及び定義済みの JAVA クラスを拡張することで) モデリング言語への機能追加が比較的容易におこなえます。Watch モデリング言語に拡張を施す方法の例として、新しい Constant オブジェクトタイプを追加します。これを、WatchApplication ダイアグラムで使用し、コードジェネレータに適用する方法を紹介します。

新しいオブジェクトタイプを追加するには、MetaEdit+ のランチャーでオブジェクトツールボタンをクリックし、'Constant' という名前のオブジェクトを定義します。Open ボタンの横のテキストボックスに未定義の名前を入力する事でオブジェクトを新規に作成できます。このオブジェクトには 'Hours' 'Minutes' 'Seconds' のプロパティを設定します (全てのプロパティのデータタイプに Number を選択します)。プロパティに設定には、Properties リストボックス上で、ポップアップメニューを開き、Add Property を選択します。表示されるリストダイアログで 'New Property Type' を選択すれば新規プロパティの作成画面が開きます。リストから既存のプロパティを選択することも可能です。表示されたプロパティウィンドウの Open ボタン横のテキストボックスに未定義の名前 (ここでは 'Hours' 'Minutes' 'Seconds') を入力する事で新規プロパティが作成されます。DataType ボタンを押して設定可能な DataType のリストを表示し、そこから Number を選択する事でプロパティのデータタイプを Number に設定できます。これらの定義を行ったオブジェクトツールは図 2 - 11 のようになります。

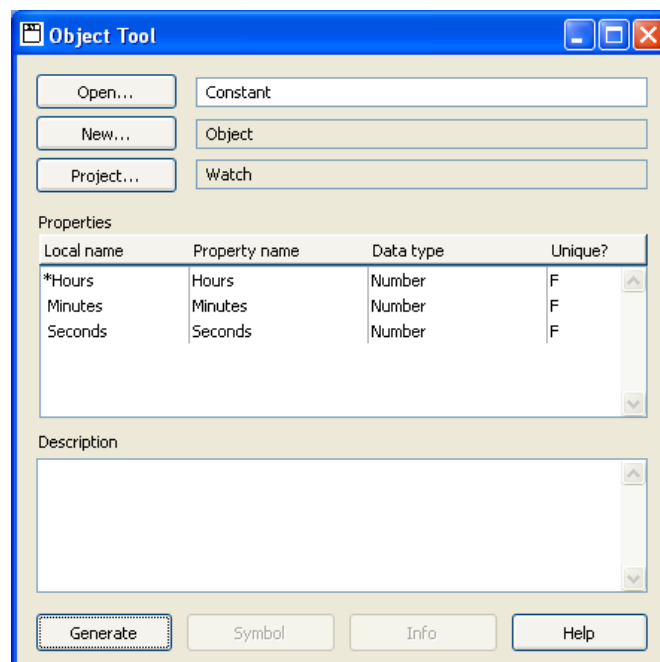


図 2-11. The Object Tool with 'Constant' definitions.

Generate ボタンをクリックして、'Constant'オブジェクトを作成します(メタモデルを始めて変更する為、この処理には数秒かかることがあります)。シンボルを作成して新しいオブジェクトタイプの定義を完了します。Symbol ボタンをクリックして、シンボリエディターを開きます。図 2 - 1 2 のようなシンボルを作成します('Variable'のシンボルをコピーして、それを修正しても構いません)。シンボリエディターを終了する前に、変更を保存することを忘れないでください。

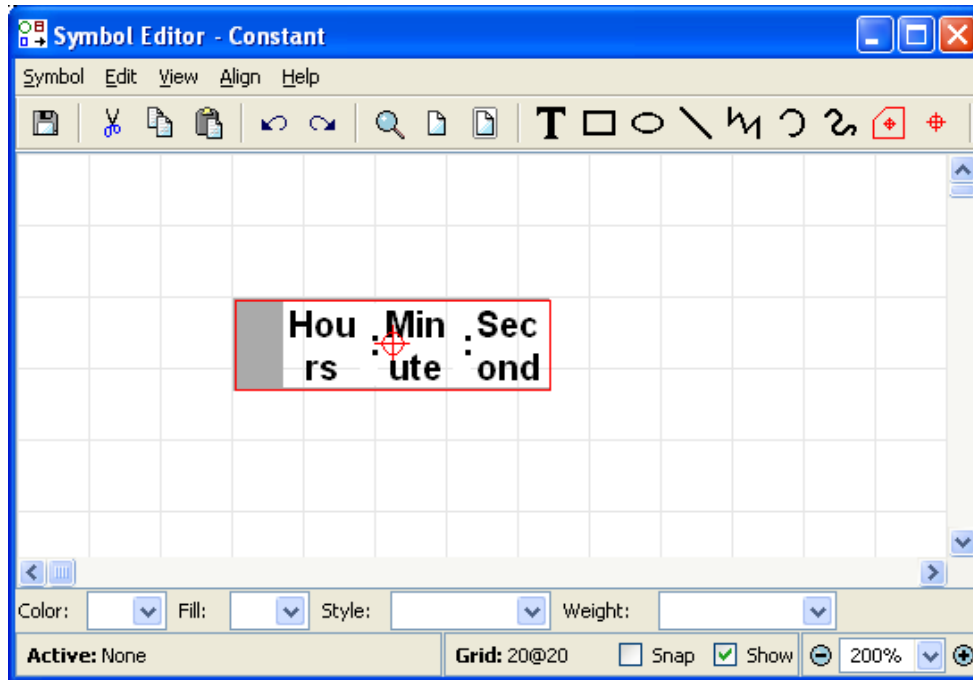


図 2-12. The symbol for 'Constant' object type.

次に、この新しいオブジェクトを既存の WatchApplication グラフタイプに統合します。MetaEdit+のランチャーからグラフツールを開き、WatchApplication グラフタイプの定義を開きます。Types ボタンを押して GraphTypes ツールを開き、Objects リストボックスに'Constant'を追加します。GraphTypes ツールを閉じて、Bindings ボタンを押し、GraphBindings ツールを開きます。'Constant'オブジェクトを'Alarm'と'Set'の全 Relationships の'Get' 'Minus' 'Plus'の Roles に追加します(Bindings は図 2 - 1 3 のようになります)。これで、'Constant'オブジェクトの作成が完了し、モデル内で使用可能になりました。

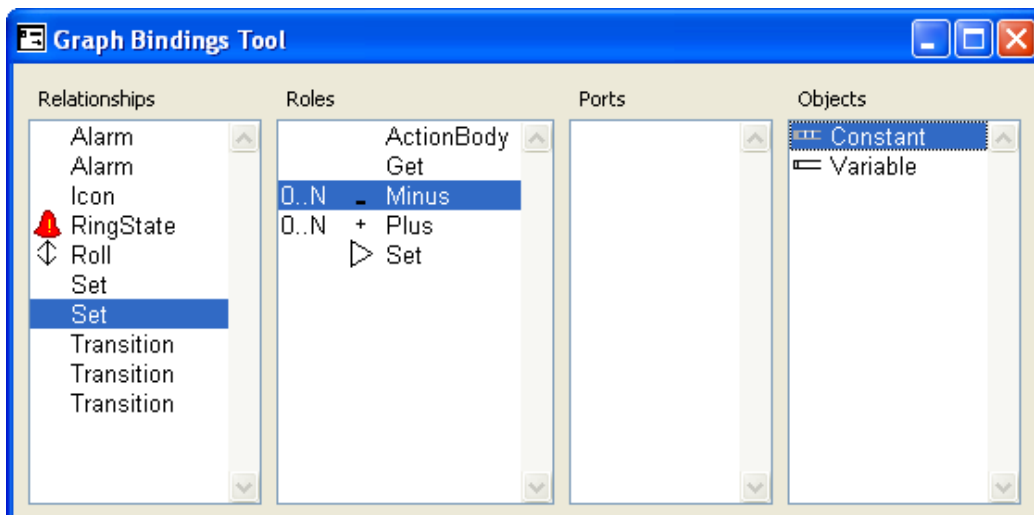


図 2-13. A binding with 'Constant' connected to 'Minus' role.

モデリング言語に新たなオブジェクトを追加する最後の工程は、コードジェネレータへの適用です。WatchApplication グラフタイプでジェネレータエディターを開き、'_calcValue'のジェネレータを以下の様に変更します。

```
Report '_calcValue'
do ~Get.()
{ if type = 'Constant'
  then '(new MTime(' :Hours ', '
    :Minutes ', ' :Seconds '))'
  else 'get' id '()'
  endif
}
do ~(Minus|Plus)
{ '.me' type
  do .()
  { if type = 'Constant'
    then '(new MTime(' :Hours ', '
      :Minutes ', ' :Seconds '))'
    else '(get' id '()'
    endif
  }
}
endreport
```

変更を反映するためにジェネレータを保存します。'Constant'タイプのオブジェクトが Watch モデリング言語のパーツとして統合されました。動作を確認するために、'Stopwatch' WatchApplication ダイアグラムの一部を検証します (図 2 - 1 4)。

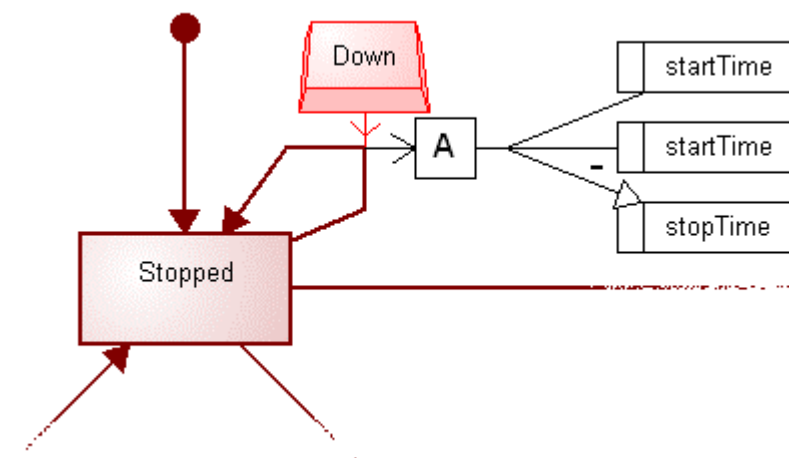


図 2-14. The original 'Stopwatch' diagram.

ストップウォッチをリセットする部分には少し野暮ったい箇所があります（'stopTime'をゼロにする為に、'startTime'の値をそれ自身から引き算し、'stopTime'に代入しています）。新しく作った'Constant'オブジェクトを使えば、'Constant'をゼロに設定して'stopTime'に代入するだけと言う、より自然な表現が使えます（図2 - 15 参照）。

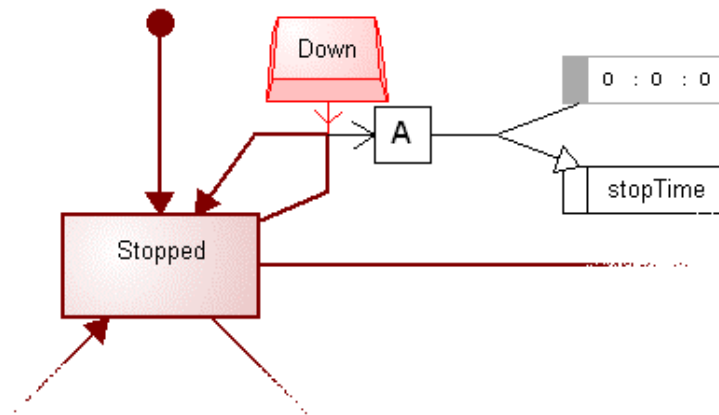


図 2-15. The new version of 'Stopwatch' diagram.

将来的な拡張に関しては、モデリング言語の実装と設計の最初の段階で、注意・認識しておく必要があります。Watch モデリング言語の実装における主要目的は、拡張の容易性ではありませんでしたが、興味深いものではありません。拡張の必要性を感じた主な部分は Action のバリエーションでした。その為、独自のリレーションシップを持った個々の動作を作ることになりました。これによって、動作に結び付くオブジェクトのルールを規定する為のバインディングや制約を簡単に作ることが出来ました。同様に、リレーションシップタイプの名前を持ったジェネレータを追加することで、この種の動作に対するコード生成も簡単に行えます。同時に、新しい動作を使ったモデリング言語の拡張が非常に簡単に行えます。

通常、拡張が複雑になればなるほどモデリング言語、コードジェネレータ及びフレームワーククラスへのさらなる修正・追加が伴います。しかしながら、これら各々がモジュール構造で実装されていると必要な修正は比較的簡単なものになります。変更を行う作業者はただ一人で十分であり、それによって、モデリング言語の利用者全てが利益を得ると言う事が最も重要な点です。

3 Watch モデリング言語をさらに詳しく見る

Watch モデリング言語を使って基本的な作業を確認してきました。ここで Watch Example をさらに詳しく見ましょう。先ず Watch Example の基本になるソフトウェアアーキテクチャーの詳細な内容を確認しましょう。

3.1 Watch アーキテクチャー

（ Watch Example のような ） DSM 環境で使うアーキテクチャーは、通常、3つの部分で構成されています（モデリング言語、コードジェネレータ、ドメインフレームワーク）。これら各々の担当範囲を理解すれば、アーキテクチャー内に於ける各々の役割も理解できます。図3 - 1 に基本原理を示します。

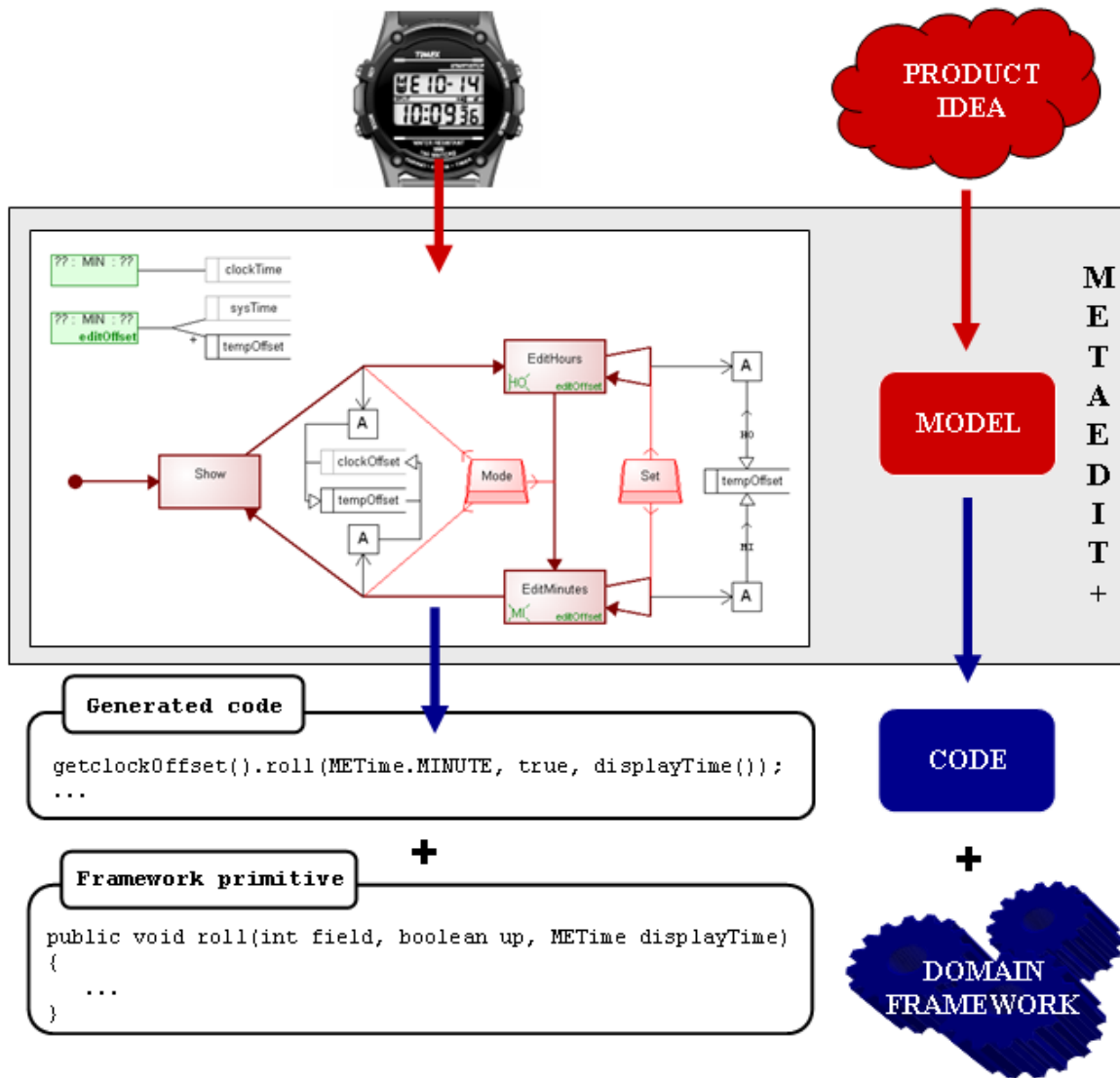


図 3-1. The watch architecture

右側にある課題を解決して、最良な抽象化レベルにする事で、Watch Example アーキテクチャーの設計と実装を行いました。先ず大まかな責任分担を見つけ出すことから始めました。この環境においてコードジェネレータが最も複雑な部分だと考えられた為、出来るだけそれを直接的且つ単純にすることに重点を置きました。この決定がモデリング言語とドメインフレームワークの役割の基礎となります。モデリング言語には製品としての腕時計とそのアプリケーションプログラムの外観と論理的な振舞を記述することを担当させることにし、コードジェネレータとのインターフェースを提供するようなドメインフレームワークを作成しました。

Watch アーキテクチャーの基本構想は合理的で単純です。以降の章で、各々のレベルで必要とされたソリューションやその実装方法について詳しく解説します。

3.2 モデリング言語

ドメイン毎に作成されたモデリング言語が、DSM 環境の最も重要な資産です。DSM アプローチによる投資回収の見込みを考察して、これを実証します。DSM 環境を構築するドメインエキスパートの労働力が初期投資です。その投資は、DSM 環境を使って働く開発者の生産性の向上結果として回収されます。ドメインフレームワークとコードジェネレータは、DSM 環境の重要な部分であるにもかかわらず開発者からはうまく隠蔽されており、モデリング言語の役割が DSM の主要な手段として強調されています。従って、良く出来たモデリング言語を使えば、DSM を使ってより多くの利益をあげることが出来ます。

ドメイン毎にモデリング言語は、モデリング言語をターゲットとなるソースコードに出来る限り依存しないように作成することが重要です。既存のソースコードやプラットフォームの拡張としてモデリング言語を構築するのは、非常に簡単な方法に思えます。しかし、この種のコードの視覚化は抽象度の向上を殆どもたらず、開発者がコードではなくドメインに基づいて考えるような環境を提供できません。従って、ドメインそのものを基準にしてモデリング言語を構築する事が、最良の抽象化レベルを得ることにつながります。

最初にドメイン固有のモデリング言語を使って何を行いたいのか？を自問しましょう。この方向性が決まれば、如何に実現するか？何がしたいか？実現するのに必要なものは何か？を問いかけることで、作業を進めて行けるでしょう。この解析が、モデリング言語の中に組み入れるべき最初のドメインコンセプトの発見を導きます。例えば、Watch モデリング言語では、方向性として次のゴールを設定しました：デジタル腕時計の構成要素になる静的なエレメントと振舞を記述できるエレメントをモデル化できるようにする。これを基により深い解析を行い、Watch Model、Display 及び Logical Watch の各コンセプトが、Watch アーキテクチャーの部品として直ぐに見つかりました。

モデリング言語を定義する上での初期段階では、ドメインコンセプトの定義と識別に重点を置かなければなりません。これには幾つかの方法があり、完璧な解答はありません。典型的に、幾つかの方針を同時に利用することが良い結果をもたらします。幾つかのケースで、ドメインエキスパートが所有しているドメインの知識が、ドメインコンセプト発見の鍵になります。

ドメインコンセプトを見つける良い方法は、ドメインとプロダクトの構造を理解することです。前に述べた Watch に於ける部品のコンセプトは、この方法で見つかりました。プロダクト内部の共通性とバリエーションを見つけるのもよい方法です。例えば、異なった腕時計に於いてボタンという共通のコンセプトを共有していると言うことを理解することは容易く、それによってボタンがモデリング言語のドメインコンセプトになりそうだと直ぐに理解できます。同様にして、腕時計のラインナップを区別する方法はボタンの数と含まれている機能であることが理解でき、そしてそれが腕時計ラインナップの構成要因であることに直ぐに結び付きます。

仕様書を精査して繰り返しパターンの有無を確かめる事で、ドメインコンセプトをより洗練されたものにする事も可能です。そして、情報源として、一般的な知恵や経験も忘れてはいけません。(組み込みシステムで一般的に広く使われている) ステートマシンの WatchApplication の基礎部分に選びました。それは、LogicalWatch 使われています。

ドメインコンセプトを定義した後は、それらをモデリング言語の中に組み込みます。モデリング言語は MetaEdit+メタモデルとして定義されます。メタモデルとは、ドメインコンセプトをモデリング言語のセマンティックスの集合体として表現したものです。メタモデル作成手順の詳細はこのチュートリアルでは触れません。メタモデル作成に関するより総合的な情報は FamilyTreeExample チュートリアルや MetaEdit+ Workbench のユーザーズガイドを参照してください。

モデリング言語の作成には、考慮すべき2つの重要な事柄があります：階層化と再利用です。全く再利用を考慮せずに、(全てのコンセプトをおなじレベルに並べる)フラットなモデル構造で、モデリング言語の開発を始めるのが典型的な例です。しかし、エレメントの数が増えてモデルが複雑になった場合に、階層化されたソフトウェアプロダクトやモジュール化されたソフトウェアプロダクトにフラットなモデルを適合させることが困難になります。そのために、階層化されたモデルを最初から提示する必要があります。

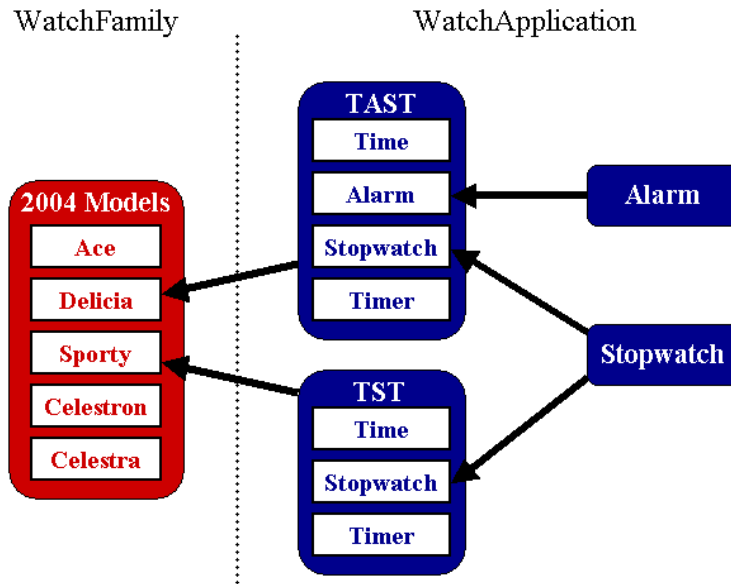


図 3-2. Watch model hierarchy

図 3 - 2 に Watch モデルの階層を図示します。Watch モデルの階層は 3 つの層で構成されています。最上位の層では WatchFamily グラフタイプを使用しており、各々の腕時計ラインナップに対する最上位の構成を規定しています。下の二つの層は、WatchApplication グラフタイプを採用しており、LogicalWatch と腕時計のアプリケーションをモデル化するために使用されます。2 つのグラフタイプを使用する事で、モデリング言語の複雑さが増しているのと同時に、それらのグラフタイプが複数の層から再帰的に利用される構造になっているので、グラフタイプを設計する為の良い練習になります。また、残り 2 つの層のセマンティクスが異なっていれば、2 つの異なったグラフタイプが必要な例になります。

再利用も階層構造に影響を与えることがあります。各々のモデルを再利用可能なコンポーネントとして扱うことが、再利用性を基本にした階層の考え方です。このような場合、再利用可能なエレメントは、必要なときは何処からでも参照でき、何処にでも保存できるように定義されています。Watch Example は、実際にはこの原理を基にして全ての階層構造を設計しています。各々の WatchApplication はどの LogicalWatch から参照可能なモデル部品です。同様に、各々の LogicalWatch はどの WatchModel にも結び付けられます。

再利用は、階層と無関係に設定することも出来ます。モデル内やモデルの間でのコンセプトの再利用が必要になることも頻繁にあります。DisplayFn のコンセプトがモデル内での再利用の例になります。WatchApplication 内で一度定義すれば、各々の DisplayFn は同じダイアグラム内の全てのステートから参照できます。モデル間の再利用は、幾つかのモデルで同じコンセプトを利用したい場合に必要になります。ボタンが Watch Example での例になります。各々のステート遷移の為に新しいボタンを作成することも可能ですが、既存のものを再利用するのがより良い方法です。この方法はボタンを手早く利用するためだけにあるのではなく、(ボタンの名前を変えるような) 将来の変更に対する伝播を簡単にするためにも使えます。この種の再利用は、エレメントを再利用するための定義が必要ないという点で、上で述べたものとは異なります。どこかで実体を作りさえすれば存在すると言う、フローティングのコンセプトとして考えられています。

最後にモデリング言語の仕上げを行います。これは一般に、コード生成、コンポーネントのインターフェース、正当性のチェック、ドキュメント生成を行う為の拡張を行うことです。例えば WatchExample では、Watch のドメインフレームワークコードを WatchFamily ダイアグラムにコンポーネントとして組み入れます。フレームワークのコードはコードを生成する時に常に利用できるように設定する必要がありました。ただなんと、これ以外にはコードやドキュメント生成の為にモデリング言語への追加や修正は必要にはなりません。

3.3 コードジェネレータ

DSM 環境におけるコードジェネレータの基本的な考え方はシンプルです：モデルを順に検索し、それらの情報を取り出し、ターゲットプラットフォームの為のコードに変換します。全ての静的・動的なロジックを記述したモデルとフレームワークが提供する低レベルのサポートを使う事で、コードジェネレータから実行可能なコードが生成されます。

MetaEdit+のジェネレータは、DSM 環境と他のツールを統合するためのツールも提供します。例えば、Watch Example で Auto-build を使えば、生成されたコードは自動的にコンパイルされ、テスト環境で実行されます。これにより、テストに於ける時間と効率の改善になります。

Watch Example では、自動ビルドの手順は以下のステップで実行されます。

- 1) **コンパイルと実行を行う為のスクリプトが生成されます。** ターゲットプラットフォームに依存した（プラットフォーム間での実行やコンパイルの自動化メカニズムの違いに対応した）スクリプトが生成されます。
- 2) **フレームワークコンポーネントのコードが生成されます。** Watch モデルには全てのフレームワークコードが含まれており、必要に応じてコードとして出力されます。この方法を使えば、全ての必要なコンポーネントが確実に利用可能で、プラットフォーム特有のコンポーネントの取捨選択も可能です。
- 3) **LogicalWatch と WatchApplication のコードを生成します。** ステートマシンは、ステートの遷移と DisplayFn のデータ構造を作成することで実装されます。 Action に対しては、それが呼び出されたときに実行するコマンドのセットをコードジェネレータは作成します。
- 4) **ターゲットプラットフォーム上のテスト環境で、生成されたコードをコンパイル・実行します。** 基本的に、このステップには最初のステップで作成されたスクリプトが必要です。

ではどのようにしてコードジェネレータが実装されているのか？ MetaEdit+におけるコードジェネレータは MERL という名称のジェネレータ定義言語で作成されています。各々のジェネレータは、特定のグラフタイプに結びついている為、特定のグラフタイプに基づいて動作します。これらのジェネレータは、サブジェネレータを呼ぶことで、階層構造にすることも出来ます。Watch Example のジェネレータアーキテクチャーの階層構造を図 3 - 3 に示します（*はターゲットプラットフォーム対応した個々のバージョンを意味します）。

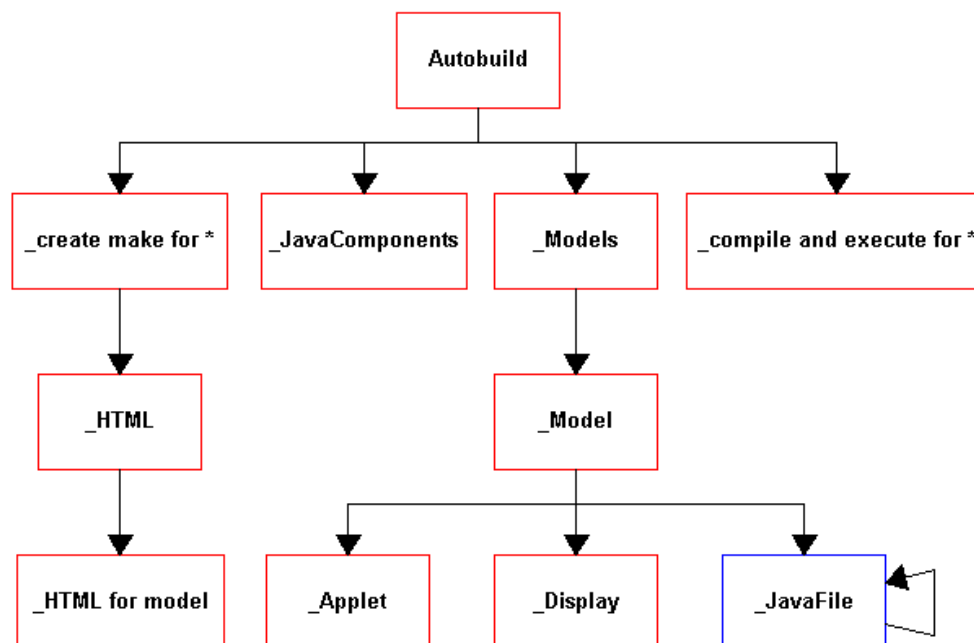


図 3-3. The watch code generator architecture, part 1

最上位に Autobuild と呼ばれるマスタージェネレータがあります。Autobuild の役割は、プログラム言語における Main プログラムと似ています：全てのジェネレーション手順を開始するけれども、基本的には下位のサブジェネレータを呼ぶだけです。Autobuild プロセスに近い次のレベルのサブジェネレータについては、この章の始めのほうで紹介します。`_JavaComponents` は事前に定義されたフレームワークの為のコードを出力するのみ、`_compile and execute *` は前のステップで作成されたスクリプトを実行するのみです。`_create make for *` と `_Models` に、より複雑なサブジェネレータの定義があります。

`_create make for *` の基本的な役割は、生成されたコードの実行とコンパイルを行う為の実行可能なスクリプトを作成することです。この手順はプラットフォームによって違いがあるため、サポートしているターゲットプラットフォーム毎に個々のバージョンが存在します。図 3 - 3 のようにブラウザベースのテスト環境のための HTML ページを必要とするプラットフォーム依存があれば、`_create make for *` サブジェネレータを使って統合することができます。

`_Models` と `_Model` サブジェネレータの役割は、Watch Model、Logical Watch、WatchApprication の為のコード生成を制御することです。各々の Watch Model には、3つのコードが生成されます：ユーザーインターフェースの物理的な実体を生成するアプレット、Watch 規定のユーザーインターフェースに関する Displayno 定義、そして、LogicalWatch。

アプレットの為に生成されたコードの例をリスト 1 に示します。アプレットを AbstractWatchApplet として定義して、クラス生成時の初期化を行っています。

```
public class Venturer extends
AbstractWatchApplet
{
    public Venturer()
    {
        master=new Master();
        master.init(this, new DisplayX334(),
            new TASTW(master));
    }
}
```

リスト 1. Generated code for an applet

Display 定義は `_Display` サブジェネレータから生成可能されます (リスト 2)。AbstractDisplay から新しい Display クラスが継承され、クラスのコンストラクターで必要なインターフェースコンポーネントが定義されています。

```
public class DisplayX334 extends
AbstractDisplay
{
    public DisplayX334()
    {
        icons.addElement(new Icon("alarm"));
        icons.addElement(new Icon("stopwatch"));
        icons.addElement(new Icon("timer"));
    }
}
```

```

times.addElement(new Zone("Zone1"));
times.addElement(new Zone("Zone2"));
times.addElement(new Zone("Zone3"));

buttons.addElement("Mode");
buttons.addElement("Set");
buttons.addElement("Up");
buttons.addElement("Down");
}
}

```

リスト 2. Generated code for a display

LogicalWatch のコード生成方法を理解する為には、コードジェネレータのアーキテクチャーをより詳しく見る必要があります。より下位のアーキテクチャーを図 3 - 4 に示します。

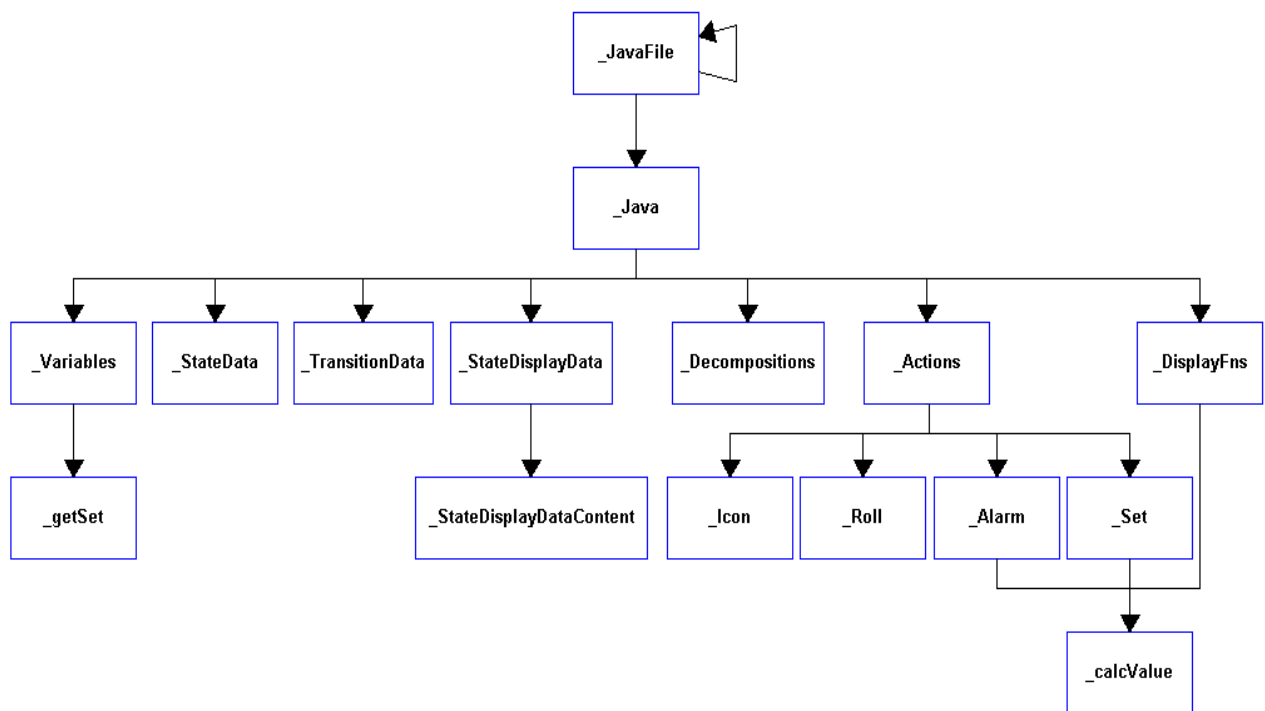


図 3-4. The watch code generator architecture, part 2

'_Java'と'_JavaFile'サブジェネレータはコード生成手順の最もクリティカルな部分を担っています：ステートマシンの実装のためのコードを生成します。他のステートマシンの中からステートマシンを呼び出すような階層構造を実現する為に、'_JavaFile'サブジェネレータは再帰構造になっています。コード生成中に、下層のステートマシンへの参照が発生したとき、'_JavaFile'サブジェネレータは下のレベルに潜り、自分自身をそこから呼び出します。

'_Java'サブジェネレータは、LogicalWatch と WatchApplication のステートマシンの最終的な JAVA 実装コードを生成します。リスト 3 に、stopwatch アプリケーションで実装されるコードの例を示します。

```
1 public class Stopwatch extends AbstractWatchApplication
2 {
3     static final int a22_3324      = +1; //+1+1+1+1
4     static final int a22_3621      = +1+1; //+1+1+1
5     static final int a22_4857      = +1+1+1; //+1+1
6     static final int d22_4302      = +1+1+1+1; //+1
7     static final int d22_5403      = +1+1+1+1+1; //
8
9     public METime startTime = new METime();
10    public METime stopTime = new METime();
11
12    public METime getstartTime()
13    {
14        return startTime;
15    }
16
17    public void setstartTime(METime t1)
18    {
19        startTime = t1;
20    }
21
22    public METime getstopTime()
23    {
24        return stopTime;
25    }
26
27    public void setstopTime(METime t1)
28    {
29        stopTime = t1;
30    }
31
32    public Stopwatch(Master master)
33    {
34        super(master, "22_1039");
35        addStateOp("Start [Watch]", "22_4743");
36        addStateOp("Running", "22_2650");
37        addStateOp("Stopped", "22_5338");
38        addStateOp("Stop [Watch]", "22_4800");
39
40        addTransition ("Stopped", "Down", a22_3324, "Stopped");
41        addTransition ("Running", "Up", a22_4857, "Stopped");
42        addTransition ("Stopped", "Up", a22_3621, "Running");
43        addTransition ("Stopped", "Mode", 0, "Stop [Watch]");
```

```

44     addTransition ("Running", "Mode", 0, "Stop [Watch]");
45     addTransition ("Start [Watch]", "", 0, "Stopped");
46
47     addStateDisplay("Running", -1, METime.SECOND, d22_5403);
48     addStateDisplay("Stopped", -1, METime.SECOND, d22_4302);
49 }
50
51 public Object perform(int methodId)
52 {
53     switch (methodId)
54     {
55         case a22_3324:
56             setstopTime(getstartTime().meMinus(getstartTime()));
57             return null;
58         case a22_3621:
59             setstartTime(getsysTime().meMinus(getstopTime()));
60             iconOn("stopwatch");
61             return null;
62         case a22_4857:
63             setstopTime(getsysTime().meMinus(getstartTime()));
64             iconOff("stopwatch");
65             return null;
66         case d22_4302:
67             return getstopTime();
68         case d22_5403:
69             return getsysTime().meMinus(getstartTime());
70     }
71     return null;
72 }
73 }

```

リスト 3. The generated code for the Stopwatch application

生成されたコードを一行ずつ確認しながら、'_Java'サブジェネレータを理解しましょう。新しい Watch アプリケーションは、AbstractWatchApplication から得られます（1行目）。生成されたコードの働きは'_Variables'、'_StateData'、'_TransitionData'、'_StateDisplayData'、'_Actions'及び'_DisplayFns'サブジェネレータで決まります。

'_Variables'と'_getSet'サブジェネレータは、後で switch-case 構造の中で使用する DisplayFn と Action のための ID を宣言します（3から7行目）。同時に、使用する変数の定義（9、10行目）と変数へのアクセスメソッドの実装（12～30行目）も行います。'_Java'サブレポートに少し戻って32～34行目が記述されています。続いて、ステート（35～38行目）とステート遷移（40～45行目）の定義が、'_StateData'と'_TransitionData'サブジェネレータによって生成されています。'_StateDisplayData'と'_StateDisplayDataContent'サブジェネレータが表示の定義を行っています（47、48行目）。そして、再度 '_Java'サブジェネレータに戻って、基本メソッ

ドの定義と switch ステートメントの開始が生成されています (5 1 ~ 5 4 行目)。

ステート遷移の間にトリガーされる Action によるコード生成 (5 5 ~ 6 5 行目) は、コードジェネレータとモデリング言語の統合方法の良い例になります。モデリング言語のレベルにおいては、各々のアクションはそのリレーションシップタイプに基づいてモデル化されます。そのコードを生成するときは、'_Actions'サブジェネレータが最初に各々の動作を定義する為の主構造を生成し、その後、アクションリレーションシップの名前 ('Icon'、'_Roll'、'_Alarm'或いは'_Set') に結びついたサブジェネレータが実行されます。この実装方法を取る事で、コードジェネレータの複雑さが軽減されるだけでなく、将来的に新しい Action が必要になったときのモデリング言語の拡張にも柔軟に対応できます。

最後に、'_DisplayFns'と'_calcValue'サブジェネレータが、表示に必要な計算を行うコードを生成しています (6 6 ~ 6 9 行目)。'_calcValue'サブジェネレータは、'_Alarm'と'_Set'サブジェネレータからも呼び出されており、コードジェネレータにおける全ての算術動作のためのテンプレートになっています。

LogicalWatch のジェネレータも同じ方法で進行します。LogicalWatch にはステート遷移に関連した Action が無い為に、生成されたコードからはうまく排除されています。さらに、より下層のステートマシーンへの参照をサポートする為に、Decomposition 構造の定義が生成されなければなりません。これは、'_Decompositions'サブジェネレータが担っています。

ここまで見てきたように、コードジェネレータに魔法があるわけではなく、ドメインに対するソリューションのモジュール化に注意して設計し、モデリング言語とドメインのフレームワークを統合しているだけです。一般的なルールは、出来るだけシンプルにコードを生成させることです：もし何か難しいことに遭遇したら、モデリング言語に上がるか、フレームワークコードに下がって考えてみましょう。

モデリング言語とコードジェネレータの説明が終わったので、ドメインフレームワークコードの説明に移りましょう。

3.4 ドメインフレームワーク

DSM の視点から見れば、ドメインフレームワークはコードジェネレータの下にある全てのもので構成されています：ハードウェア、OS、プログラミング言語及びソフトウェアツール、ライブラリと全ての追加コンポーネント。しかしながら、ドメイン固有の部分と一般的なプラットフォーム依存の部分にフレームワークを分解する事で、完璧な DSM 環境を作るのに適したフレームワークの要求を理解できます。

殆んどの場合、プラットフォームとフレームワークのドメイン固有部分との境界は明確ではありません。例えば、Watch Example を最初に書いた時の JAVA のバージョンでは、アラームのような時間に依存したイベントを扱う有用なサービスがありませんでした。従って、ドメインフレームワークの一部として、そのサービスを独自に作成しました。最新の JAVA のバージョンでは、同様の機構が提供される為、それがプラットフォームの一部になりました。

フレームワークとプラットフォームの境界に関して、理論的な議論を避けると以下の定義が利用できます：プラットフォームはハードウェア、OS、JAVA プログラミング言語 (AWT クラスを含む) 及び生成されたコードをテストする環境を含むと考えます。ドメインフレームワークは、このプラットフォームの上であり、コード生成を支援する為に必要な追加のコンポーネントやコードと考えます。この方法で定義した、Watch ドメインのアーキテクチャーを図 3 - 5 に示します (実線の矢印はインスタンスを表し、点線の矢印は包括を表します)。

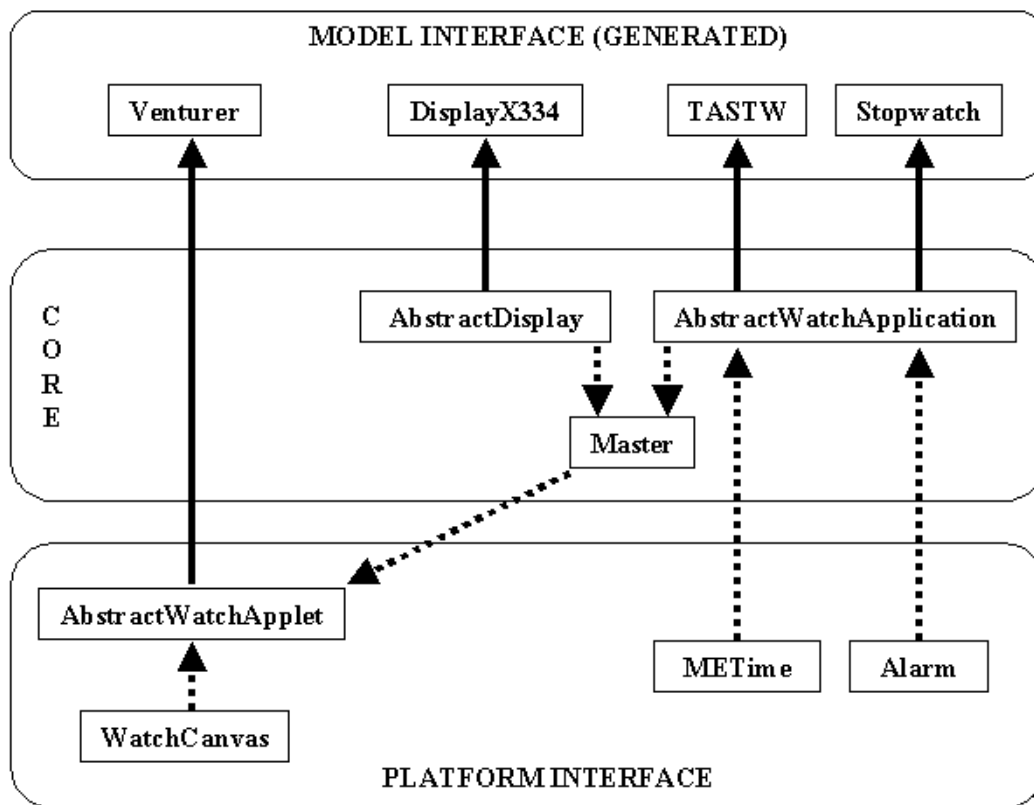


図 3-5. The watch domain framework

Watch Example のドメインアーキテクチャは3つのレベルで構成されています。最下層はターゲットプラットフォームとのインターフェースに必要な JAVA クラスです。中間層はフレームワークのコアで、Abstract Superclass テンプレートの形式で、Watch モデルの基本ブロックを提供しています。最上位層はモデルとフレームワークのインターフェースを提供します。この層で生成されたコードは他の層で提供されるコードやテンプレートと一緒にコンパイルされる様に定義されます。

フレームワークの最上位には2種類のクラスがあります。METime と Alarm は、プラットフォームの複雑さを隠蔽し、コードレベルでの抽象度を上げるために作成しました。例えば、アラームサービスの実装は、かなり複雑な JAVA の実装を利用しています。この複雑さを隠蔽するために、Alarm クラスには、アラームのセットと停止を行う為の単純なサービスのインターフェースを実装しました。同様の手順で、METime は JAVA の日付と時間に関する実装の欠点を補っています。コード生成を行うと、アラームのセットや時間の計算が必要なときに、これら2つのクラスによって提供されるサービスを呼び出す単純な呼出し命令が生成されます。

下位層の AbstractWatchApplet と WatchCanvas クラスは、プラットフォーム依存のユーザーインターフェースから Watch アーキテクチャを分離する重要なメカニズムを提供します。サポートされる各々のターゲットプラットフォームに対応した個々のバージョンが用意され、コードジェネレータによって、必要とするインターフェースを実現するたった一つのターゲットテンプレートのコードだけが確実に選択されます。

プラットフォームの上にはフレームワークのコアがあります。インターフェースとそのサービスの利用手段を提供します。コアはモデルによって提供されるロジック構造の対象物を実装します。WatchApplication、LogicalWatch 及び Display の Abstract 定義がここにあります (AbstractWatchApplication と AbstractDisplay クラス)。コードジェネレータがモデル内にこれらの定義を見つけたとき、対応する Abstract クラスのサブクラスを作成します。

プラットフォームインターフェースやコードレベルと違って、モデルインターフェースレベルには、事前に定義されたクラスが全く含まれていません。それどころか、AbstractWatchApplet、AbstractWatchApplication 或いは AbstractDisplay のサブクラスが作られたときにジェネレータが出力しなければならない API やルールのセットのようなものになっています。

典型的に、フレームワークは主として内製のコンポートで構成されており、前のプロジェクトで既に利用されていたり、この目的の為に作成されていたりします。

<お問い合わせ・資料請求>

富士設備工業株式会社 電子機器事業部

E-MAIL: info@fuji-setsu.co.jp

www.fuji-setsu.co.jp

