コンポーネントウェアを用いた自動販売機制御ソフトウェアの作成支援に関する研究

97B506 別府 麻美　97B517 橋口 博文
97B524 市村 豪規

指導教員　野呂 昌満

1　はじめに

富士電機では，自動販売機ソフトウェアの一部をエンドユーザに開放し，エンドユーザ自身で要求を満たすソフトウェアを作成するエンドユーザコンピューティングに移行する計画がある．エンドユーザとは，コカコーラ，ポッカなどの商品を提供する側のことをさす．エンドユーザコンピューティングへの移行のさい，以下の制約がある．

- 既存のコードの再利用
- エンドユーザに使用者定義コンポーネントとして，ソフトウェアの一部を開放する
- オブジェクト指向言語 Java をもちいる

エンドユーザに開発環境を提供することで，以下の利点がある．

- 開発側 ソフトウェアの再利用性の向上
- エンドユーザ側　コードを意識することなくソフトウェアを作成できる

エンドユーザに開放する自動販売機ソフトウェアの部分は，エンドユーザによって異なる自動販売機制御ソフトウェアのコントロール以外の部分である．エンドユーザコンピューティングに移行するさい，以下の問題点があげられる．

- 分析レベルの文書が欠落しているので，コンポーネントとして提供する部分がわからない
- 疑似並行実行の実現方法が複雑であるので，コンポーネント化が困難

上記の問題を解決するために，エンドユーザコンピューティングへの移行手順の確立として，

1. デザインリカバリ
2. 疑似並行実行の実現方法の改善
3. 使用者定義コンポーネントの提供

をおこなう．

エンドユーザコンピューティングへの移行手順の確立を最終目標として，

- 疑似並行実行単位のコンポーネント化
- 使用者定義コンポーネントの定義
- エンドユーザコンピューティングへの移行手順に対する妥当性の検証

をおこなうことを本研究の目的とする．

エンドユーザがコードを意識することなく意図する実行の単位を作成できるよう実行の単位をコンポーネント化する．デザインリカバリしたモデルからコンポーネントの作成手順を提示し，例をもちいて作成手順にそってコンポーネントが作成できるか確認する．

自動販売機シミュレータを作成し，通信部分を確認することにより，最終目標であるエンドユーザコンピューティングへの移行手順が，妥当かを検証する．

2　改善提案の実現方法

改善提案を実現するための手順を以下に説明する．

2.1　デザインリカバリ

開発したソフトウェアの文書を修復するために，デザインリカバリをおこなう．以下の手順でデザインリカバリをおこない，分析レベルの文書を修復し，モデルを定義する．

1. オブジェクト名に着目
2. オブジェクト名をもとに分類
3. 分類したオブジェクトの機能に着目
4. 機能をもとにグループ化
5. グループ化したオブジェクト群の関係を線で結ぶ

本研究では自動販売機制御ソフトウェアの一部であるテンキーのデザインリカバリをおこなった．

2.2　疑似並行実行

現在の自動販売機制御ソフトウェアはコールバック関数を疑似並行実行の単位として考え，疑似並行実行を実現している．コールバック関数には，疑似並行実行を設定するものと実行するものが存在し，一本の実行経路を複数に分割するので，複雑な実行経路になり分かりにくい．疑似並行実行の実現方法の改善として，Command パターン [?] を利用し疑似並行実行を実現する．コマンドオブジェクトを実行の単位とし，疑似並行実行を実現すると一本の実行経路を分割することなく疑似並行実行を実現できる．

2.3　使用者定義コンポーネント

エンドユーザがコードを意識することなく，ソフトウェアを作成するために使用者定義コンポーネントを提供する．開発側はコンポーネントをもちいることでソフトウェアの再利用性の向上が期待できる．

本研究では疑似並行実行の実行単位である Command と，自動販売機制御ソフトウェアの一部であるテンキーをコンポーネントとしてどのように定義するか考える．

3　疑似並行実行の実現方法の改善

Command パターンを利用し，コマンドオブジェクトを実行の単位とし，疑似並行実行を実現するさい，コマンドを管理するモニタが必要となる．モニタのインタフェースを考え，モニタの設計，必要となるオブジェクトを考える．

3.1 疑似並行実行単位

疑似並行実行の実現方法の改善をおこなうさい，必要となるオブジェクトを以下に説明する.

task モニタが管理する実行の単位であり,primitive_task か composite_task である.

primitive_task Command パターンを使った実行の最小単位であるオブジェクト.

composite_task task の集まりからなり, 内部状態をもつオブジェクト.

state composite_task の内部状態を表すオブジェクト.

event ある事象を表すオブジェクト.

condition event_condition か simple_condition か not_condition か or_condition か and_condition である.

event_condition ある一つの event が起こったかどうか判断し, 結果を他のオブジェクトに教えるオブジェクト.

simple_condition 一つの条件を持っており, 条件が成り立っているか判断し, 結果を他のオブジェクトに教えるオブジェクト.

not_condition 一つの condition を持ち,condition の条件が not か判断し, 結果を他のオブジェクトに教えるオブジェクト.

or_condition 二つの condition を持ち, 二つの condition の条件が or か判断し, 結果を他のオブジェクトに教えるオブジェクト.

and_condition 二つの condition を持ち, 二つの condition の条件が and か判断し, 結果を他のオブジェクトに教えるオブジェクト.

guarded_task モニタの登録単位であり, 一つの condition と一つの task を持ち,condition のもつ条件が成り立ったとき,guarded_task が持つ task が実行可能であることをモニタに教えるオブジェクト.

以上のオブジェクトの関係を表すクラス図 [?](図??) を示す.

composite_task がもつ内部状態が変化したときに composite_task の振る舞いを変えるために,State パターン [?] の利用が考えられた.

3.2 モニタの設計指針

モニタは event を監視し， task を保持，管理する． task の実行は全てモニタがおこなうことで， task 同士の同期をとる．また，モニタを設計するにあたり二つのことを考え, 実現する必要があった.

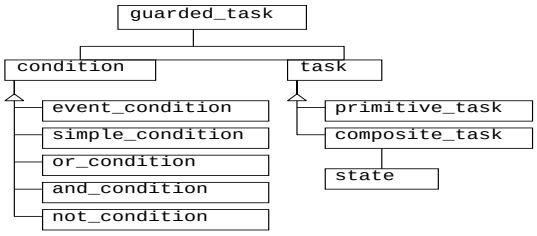

図 1: task state event condition

- selective_wait

  event により, 複数の task から一つの実行可能な task が選択され実行される．実行されなかった task はモニタの登録から削除される.
- concatenation

  複数の event が決められた順に起こることによって composite_task の状態遷移が起こる.

以上の設計指針にもとずき設計されたモニタのインターフェースを以下に説明する.

モニタのインターフェース

register 引数でもらった guarded_task を登録するインタフェース.

set_wait selective_wait の組を表す識別子である wait_tag を返すインタフェース.

set_subtask 引数でもらった wait_tag が表す selective_wait の組として guarded_task を登録するインタフェース.

complete_wait 引数でもらった wait_tag が表す selective_wait の組の登録が終わったときに呼ばれるインタフェース.

set_concate concatenation の組を表す識別子である concate_tag を返すインタフェース.

set_concate_subtask 引数でもらった concate_tag が表す concatenation の組として guarded_task を登録するインタフェース.

complete_concate 引数でもらった concate_tag が表す concatenation の組の登録が終わったときに呼ばれるインタフェース.

cancel 引数でもらった task を登録されている task から削除するインタフェース.

execute 起きた event を引数でもらい,

1. 起きた event によりモニタに登録されている task の中から実行可能な task を探す.

2. 実行可能な task を実行する.

以上のモニタの設計により Command パターンを使った疑似並行実行を実現する.

3.3 疑似並行実行単位のコンポーネント化

エンドユーザがコードを意識することなく意図する task 作成できるように task, state, condition, guarded_task を JavaBeans をもちいてコンポーネント化した. state をコンポーネント化するさい, 4 種類の state コンポーネントを作成した.

selective_state コンポーネント 遷移する次の状態が複数ある state コンポーネント.

concatenation_state コンポーネント concatenation を実現する state コンポーネント.

sequence_state コンポーネント 遷移する次の状態が一つある state コンポーネント.

end_state コンポーネント 終了状態を表す state コンポーネント.

図 2 の状態遷移をする task(a_composite_task) を例にあげ, task を JavaBeans 上で作成する手順を説明する.

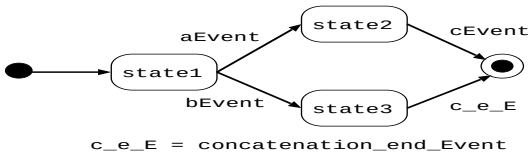

図 2: a_composite_task の状態遷移図

a_composite_task は,

state1 のとき モニタ上で実行されると guarded_task1, guarded_task2 をモニタへ登録する. 登録後, aEvent が発生したとき guarded_task1 がもつ primitive_task1 がモニタ上で実行される. bEvent が発生したとき guard- ed_task2 がもつ primitive_task2 がモニタ上で実行される. state1 から次の状態へ移行するさい, 実行されなかった guarded_task1 か guarded_task2 どちらかをモニタの登録から削除する.

state2 のとき モニタ上で実行されると guarded_task3 をモニタへ登録する. 登録後,cEvent が発生したとき guarded_task3 がもつ primitive_task3 がモニタ上で実行される.

state3 のとき モニタ上で実行されると guarded_task4, guarded_task5 をモニタへ登録する. 登録後, dEvent が発生したとき guarded_task4 がもつ primitive_task4 がモニタ上で実行される.dEvent の発生後, eEvent が発生したとき guarded_task5 がもつ primitive_task5 がモニタ上で実行される.

終了状態のとき モニタ上で実行されると自分自身をモニタの登録から削除する.

作成手順

1. 各 state コンポーネントと composite_task コンポーネントを BeanBox[?] に張り付ける. state1 は selective_state コンポーネントであり, state2 は sequence_state コンポーネントである. state3 は concatenation_state コンポーネントである.
2. 各状態に次の状態を登録するために状態遷移表とは逆に線を結び, プロパティシート [?] 上で各状態の名前を登録する.
3. 各状態でモニタに登録する guarded_task コンポーネントを作成し, composite_task コンポーネントと意図する state コンポーネントへ guarded_task コンポーネントを登録するために線を結ぶ.concatenation_state コンポーネントに guarded_task コンポーネントを登録する場合は, 実行してほしい順番で線を結ぶ. guarded_task コンポーネントを作成する手順は, guarded_task コンポーネントと意図する condition コンポーネントと task コンポーネントを BeanBox 上で張り付け condition コンポーネントと task コンポーネントを guarded_task コンポーネントに登録するために線を結ぶ.
4. 各 state がもつ次の状態へ遷移する条件 (Event) をプロパティシート上で登録する. selective_state コンポーネントは対応した条件と次の状態の名前を登録する. concatenation_state は次の状態へ遷移する条件 (Event) は, モニタが発生する concatenation_end_Event であるので, 次の状態へ遷移する条件 (Event) は登録できない. 手順 1,2,3,4 後の図を (図??) に示す.

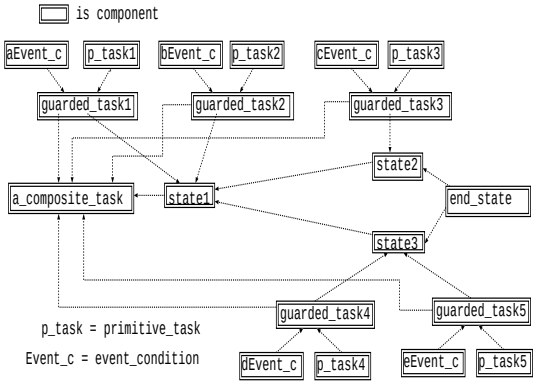

図 3: 手順 1, 2, 3, 4 後の図

以上のようにエンドユーザはコードを意識することなく, 簡単に意図する task をコンポーネントの組合せで作成できる.

4 使用者定義コンポーネントの定義

デザインリカバリしたモデルからコンポーネントを作成する手順を提示する. 提示した作成手順にそって実例をもちいてコンポーネントを作成することで作成手順が妥当か評価する.

作成手順

1. 分析レベルの文書の各クラスの役割をきめる
2. ある処理をおこなうさい, クラス間のメッセージのやりとりに着目する
3. 一つのクラスを複数のコンポーネントに分解できるか判断する
4. 複数のクラスを一つのコンポーネントにまとめることができるか判断する
5. クラスとコンポーネントを同じとみなしてよいか判断する
6. 各コンポーネントのインタフェースを定義する

手順 1 から手順 6 にしたがって, 例として本研究でデザインリカバリしたテンキーのモデル (図??) をもとにコンポーネントを作成する.

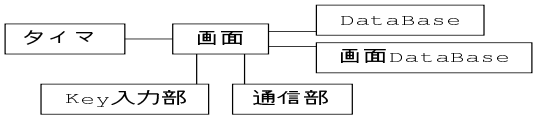

図 4: テンキーのモデル

4.1 テンキーのコンポーネント化

テンキーのコンポーネントを作成するにあたって以下の制約がある.

- エンドユーザが task, モニタ,event,state を意識しないでコンポーネントをあつかうことができる
- 作成したコンポーネントはモニタ上で動作する
- 開発環境は JavaBeans をもちいる

設計

テンキーのコンポーネントを作成手順にそって作成する.

手順 1 でモデルの各クラスの役割をきめた.

- Key 入力部 : テンキーを操作するボタンをもつ
- タイマ : 入力操作の時間切れを管理する
- 通信部 : 自動販売機との通信をおこなう
- DataBase : 商品の価格, 商品の売上額, 商品の売上数をもつ
- 画面 DataBase : 画面に表示するメッセージをもつ
- 画面 : 入力値の表示, 処理の結果を表示する, 他クラスとのメッセージのやりとりを管理

手順 2 ではテンキーがおこなう処理におけるクラス間のメッセージのやりとりを確認する. 本研究で作成するテンキーは商品の価格設定, 商品の売上額確認, 商品の売上数確認, つり銭の補充, 商品の補充をおこなうことができる. 自動販売機から通信部へ通知される情報はテンキー操作可能, 商品の販売とする. 各処理のやりとりをシーケンス図をもちいてまとめた.

手順 5 までででクラスからコンポーネントの分類をおこなう. コンポーネントに分類するさい, コンポーネント間の関係が複雑にならないようにするために, あるクラスで, メッセージのやりとりをするクラスが限られている場合, 二つのクラスを一つのコンポーネントとしてまとめた. テンキーでは画面, 画面 DataBase および画面,DataBase のメッセージのやりとりが該当するので, 画面,DataBase, 画面 DataBase を一つのコンポーネント (ScreenUnit) として考えた. テンキーは以下の 4 つのコンポーネントに分類できた.

- ScreenUnit
- Key 入力部 (KeyBoard)
- タイマ (Timer)
- 通信部 (Communication)

テンキーのコンポーネントの分類結果 (図??) を示す.

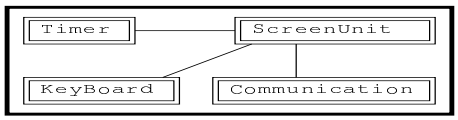

図 5: コンポーネントの分類結果

手順 6 では, 手順 2 で示した処理をおこなうさい, コンポーネント間のメッセージのやりとりを確認してインタフェースを定義する.

作成するコンポーネントがモニタ上で動作するには,

- 処理を記述した task
- task を実行するきっかけとなる event
- モニタへ task と event を登録するために必要となる guarded_task

を必要に応じてコンポーネントにもたせる. 作成するコンポーネントは次の手順でモニタ上で動作する.

1. guarded_task をモニタに登録
2. 発生した event に対応した task をモニタが実行
3. task に記述した処理を実行

task は処理が記述されているコンポーネントを知る必要がある. task を生成するさい, 実行先のコンポーネントを

教えることで task はモニタに呼び出されると処理が記述されているコンポーネントのインタフェースを呼び出すことができる.

KeyBoard の数字の 1 ボタンを押して ScreenUnit の画面に表示するまでの処理の流れを示す (図??).

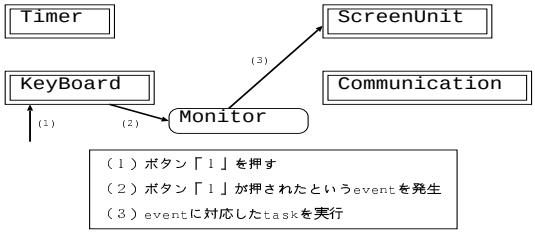

図 6: 処理の流れ

各処理のコンポーネント間のやりとりから必要な task, event, guraded_task を用意してモニタに event を通知することで, 実行先のコンポーネントのインタフェースを呼び出すことができる.

実現

設計したコンポーネントのインタフェースをもとに JavaBeans をもちいて実現した. コンポーネント間の通信はコンポーネントをはりつけるだけで通信可能となる.task を生成するときに実行先のコンポーネントを教えることでモニタに呼び出された task から生成したコンポーネントの処理を呼び出すことができる.

4.2 作成手順の評価

テンキーのコンポーネントを作成することで, 提示した作成手順が妥当か評価する.

確認事項を以下に示す.

- 作成したコンポーネントはモニタ上で動作する
- エンドユーザは task, モニタ,event,state を意識しないでコンポーネントをあつかうことができる

コンポーネント間で一つの処理をおこなうさい,

- 処理を記述した task
- task を実行するきっかけとなる event
- モニタへ task と event を登録するための guarded_task

を用意することで, コンポーネント間のメッセージのやりとりはモニタを介しておこなうことになる. 作成したコンポーネントはモニタ上で動作する.

作成したコンポーネントは BeanBox にはりつけるだけで操作できる. 提供するコンポーネントの数は少ないので, エンドユーザにとってソフトウェアを作成するさい, あつかいやすいものとなった.

提示した作成手順にしたがうことでデザインリカバリしたモデルからモニタ上で動作するコンポーネントを作成できる.

5 改善案に対する妥当性の検証

コンポーネント化したテンキーと自動販売機本体が, 正常に通信をおこなっているかを確認するために, 自動販売機シミュレータを作成する.

5.1 自動販売機シミュレータの再設計

過去に作成された自動販売機シミュレータ vend7[?] には, 以下の問題点があげられる.

- テンキーの機能をいろいろなオブジェクトが保持している
- 機能中心にオブジェクトを定義しているので, 不必要なオブジェクトが存在する

上記の問題を解決するために, モデルを再定義し, 共通した機能をもつオブジェクトを一つにまとめて, 不必要なオブジェクトをなくした.

ソフトウェアの参照アーキテクチャ

ソフトウェアの参照アーキテクチャとは, ある問題を取りあつかうとき, 問題領域に共通して当てはまるようなソフトウェアのおおまかな構成を示すものである. 参照アーキテクチャをもちいることで, 共通した機能をもつオブジェクトをグループ化することができるので, ソフトウェア内に存在する機能の独立性を図ることができる.

三層モデル

過去の研究 [?] で定義された自動販売機シミュレータの参照アーキテクチャを,

- 状態遷移層
- コントローラ層
- ハードウェアモデル層

の三層モデルで再定義した (図??).

状態遷移層 event によって内部の状態を遷移させることで, 自動販売機シミュレータの機能を実現している層である. 入力されたデータに対して処理をおこない, その結果をコントローラ層のオブジェクトに伝える. 状態遷移層のオブジェクトは, 次の 4 つである.

- 金銭の管理をおこなうオブジェクト
- 商品の管理をおこなうオブジェクト
- 現在金額の状態を管理するオブジェクト
- 自動販売機の状態を管理するオブジェクト

テンキーを含めた自動販売機内部の状態遷移図を示す (図??).

コントローラ層 自動販売機内に存在する仮想的なハードウェアをモデル化したものを配置する層である. 状態遷移層とハードウェアモデル層の間にコントローラ

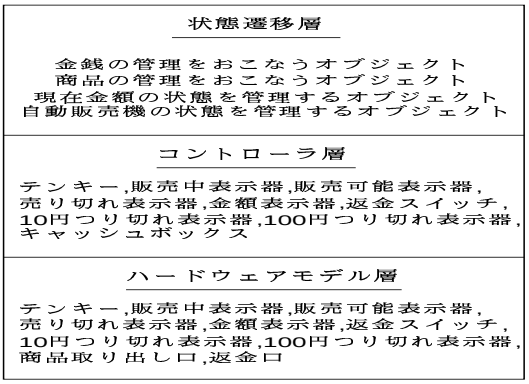

図 7: 三層モデル

層が存在することによって, 実際のハードウェアへの出力と状態遷移層への入力を考える必要がなくなる.

ハードウェアモデル層 実際の自動販売機を構成するハードウェア部品をモデル化したものを配置する層である.

5.2 自動販売機シミュレータの試作

以下の手順で自動販売機シミュレータを作成した.

- 再定義した三層モデルをもとに, オブジェクトを作成する
- テンキーと自動販売機本体の通信部分となるクラス (Communication) のインタフェースを定義する

Communication のインタフェースには, 商品の補充, 価格の通知, つり銭の補充, 売上の通知, 自動販売機の状態の通知がある. テンキーと自動販売機本体は,Communication を介してやりとりする. Communication は自動販売機本体のオブジェクトとテンキーの ScreenUnit を知っている.

5.3 妥当性の検証

JavaBeans 上で作成した task をモニタに登録し, 疑似並行実行をしているか確認した. JavaBeans 上で Communication と自動販売機シミュレータが正常に通信していることを確認した. 最後に, 価格設定の task を生成し, 価格設定をおこなった結果, エンドユーザコンピューティングへの移行手順の確立の見通しを得た.

6 おわりに

エンドユーザが意図した task を簡単に作成できるように疑似並行実行単位をコンポーネント化した. 使用者定義コンポーネントの作成手順を提示しテンキーのコンポーネントを作成した. 再定義した三層モデルをもとに, 自動販売機シミュレータを作成し, 妥当性の検証をおこなった結果,

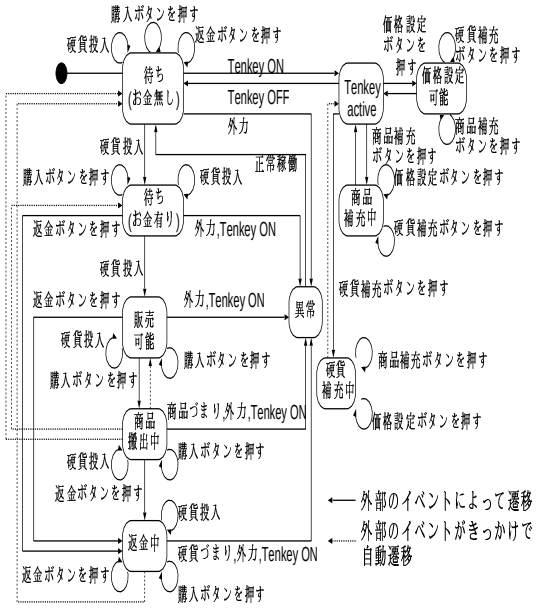

図 8: 自動販売機内部の状態遷移図

エンドユーザコンピューティングへの移行手順の確立の見通しを得た.

今後の課題は,

- 例としてもちいた TenkeyUnit 以外の場合での妥当性の検証
- 操作性にすぐれたコンポーネントの再定義

である.

参 考 文 献

[1] Brookshier Dan, “JavaBeans デベロッパーズリファレンス”, ソフトバンク,1997.

[2] Erich Gamma, et. al, “Design Patterns”,Addison Wesley,1995.

[3] Eriksson,Hans E,Penker,Magnus,“UML Tool Kit”, トッパン,1999.

[4] 栗田浩和, 野崎奈穂,:, 自動販売機制御ソフトウェアの構成法に関する研究, 1997.