データベースアプリケーション構築技法

Delphi、C++Builderによるメンテナンス性を考慮した開発

第二章：実際のアプリケーション開発の指針（基礎）

Borland®

Client/Server システムの開発においての指針

- アーキテクチャを正しく理解する
  - データベース（DBMS）を正しく利用する
  - クラスライブラリ、フレームワークを正しく利用する
    - VCL フレームワークの理解と正しい利用
- 設計
  - アプリケーションに必要な機能のみを実装する
    - その処理は本当に必要か？
      - クライアントで処理するものか？DBMS側で処理するものか？
  - オブジェクト指向、コンポーネント指向の有効活用
    - ソースコードを役割毎に正しく分割し、実装する

データベース（DBMS）を正しく利用する

- データベース構造を正しく正規化する
  - 参照が主なら一部を非正規化してみる
- RDBMSのデータ処理能力を活用する
  - サーバーで提供される機能を活用する
    - ストアドプロシージャ、ビュー等の活用
      - 不用意にクライアントに処理を預けてはいけない
    - NULL(ナル)値を正しく使う
      - NULLが不要な項目に、NOT NULL制約を必ず付ける
  - インデックスを不必要に設定しない
    - インディックスを正しく設定する
    - インデックスは設定しない方がいい項目型がある
  - チューニング
    - ハード、ソフトの適切なチューニングを行う
    - SQL文をチューニングする

クラスライブラリ、フレームワークを正しく利用する

- VCL の各コンポーネントには、それぞれ役割がある
  - VCL フレームワークの理解と正しい利用が肝心
- 安易に VCL コンポーネントの機能を用いない
  - VCL の機能はどのように実現されているかを理解しないとパフォーマンスが落ちる
    - 基本的に 接続先が DBMS の場合、Table タイプのコンポーネントは用いない
      - Query タイプのコンポーネントで処理を行う
      - Table タイプのコンポーネントでは、細かいデータ処理をコントロールできない
    - 過度な UI を提供するコンポーネントを多用しない
      - 参照項目や Lookup など
      - 手軽な操作で大量レコードが必要になる
    - Filter プロパティなど、クライアントにレコード処理をさせる実装をなるべく利用しない
      - フィルタを行うために内部でデータが呼ばれ、必要のないデータがネットワークを流れる
      - この場合、SQL 文で 処理を行う (Select a,b,c form table where …)
    - RecordCount プロパティは使わない。
      - レコード数を取得するためにトラフィックが発生
      - レコード数を取得は「SELECT COUNT(*)」文でチェックする
      - 結果セットのレコードの有無は、Eof プロパティでチェックする

Borland®

アプリケーションに必要な機能のみを実装する

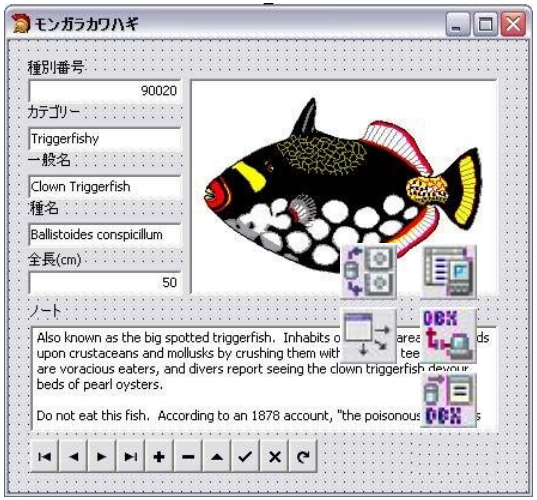
- ネットワーク上に流れるデータ量を可能な限り少なくする
  - SQL を積極利用し必要なデータのみ抽出する
  - 余分なデータをネットワークに流さない
    - そのデータが本当に必要かを考える
      - 数百から数千件のデータをクライアントに送るような処理
        - これ以上の規模のデータをクライアントに送っても人手での処理は難しい
        - レポートやグラフ化などは集計値のみを利用できるか検討する
    - 大量のSQL文を発行する処理
      - ストアド・プロシージャで実装できないか検討する

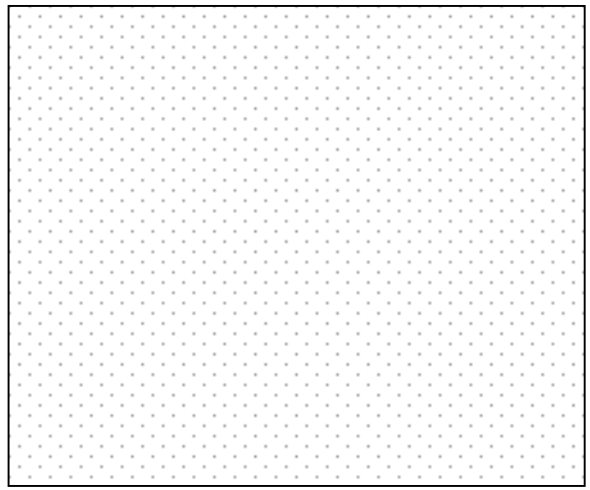
オブジェクト指向、コンポーネント指向の有効活用

- ソースコードを役割毎に正しく分割し、実装する
  - Delphi / VCL のアーキテクチャを理解した上での設計
    - UI や出力系のロジックとデータ操作ロジックを密接に連携させてはいけない
      - データモジュールの有効活用
        - 機能を役割毎に正しく分割でき、かつビジュアルに機能を実装できる
      - UI や出力系のロジックと データモジュール間のでやり取りを行なう
        - なるべく UI(Form)内からデータ操作コンポーネントを呼ばない
        - データ操作時の例外処理等を UI 内で処理しない
        - データモジュールに適切な操作関数などを用意し UI から呼ぶようにする

Borland®

データモジュールを積極的に利用する

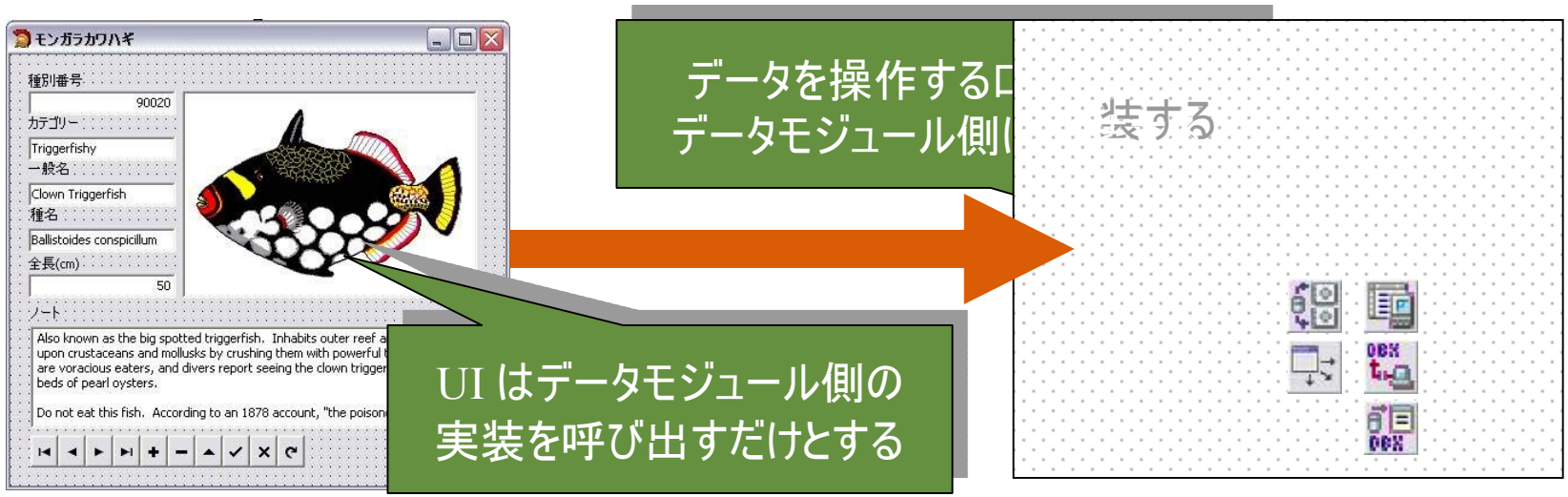




- UI や出力系のロジックとデータ操作ロジックを密接に連携させてはいけない
  - データ操作ロジックの再利用やメンテナンス性を確保できなくなる
  - データモジュールを積極的に利用し、ロジックを実装していく
    - 可能な限り、Form にデータセットコンポーネントを貼らない
      - データ操作 / ビジネスロジックは、データモジュールに集める
    - データモジュールに、適切なデータ操作のための実装を作成する
      - Public 関数などを利用
      - 例外などのエラー処理を含め、データモジュール内にデータ操作コードを完結した形で実装する
        - 特定のフレームワークに依存した例外をアプリケーション独自の例外などで処理するのも有効

Borland®

データモジュールを積極的に利用する

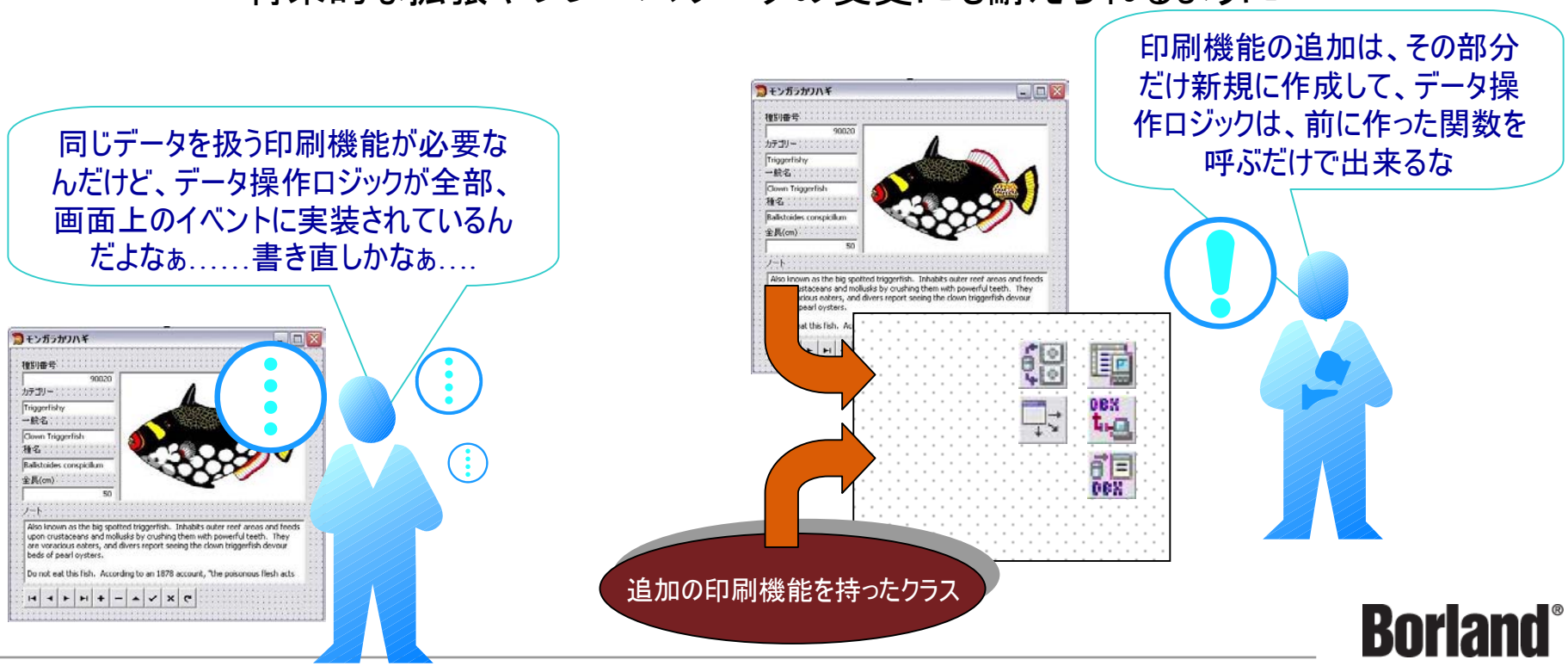


- UI や出力系のロジックとデータ操作ロジックを密接に連携させてはいけない
  - データ操作ロジックの再利用やメンテナンス性を確保できなくなる
  - データモジュールを積極的に利用し、ロジックを実装していく
    - 可能な限り、Form にデータセットコンポーネントを貼らない
      - データ操作 / ビジネスロジックは、データモジュールに集める
    - データモジュールに、適切なデータ操作のための実装を作成する
      - Public 関数などを利用
      - 例外などのエラー処理を含め、データモジュール内にデータ操作コードを完結した形で実装する
        - 特定のフレームワークに依存した例外をアプリケーション独自の例外などで処理するのも有効

Borland®

データモジュールを積極的に利用する

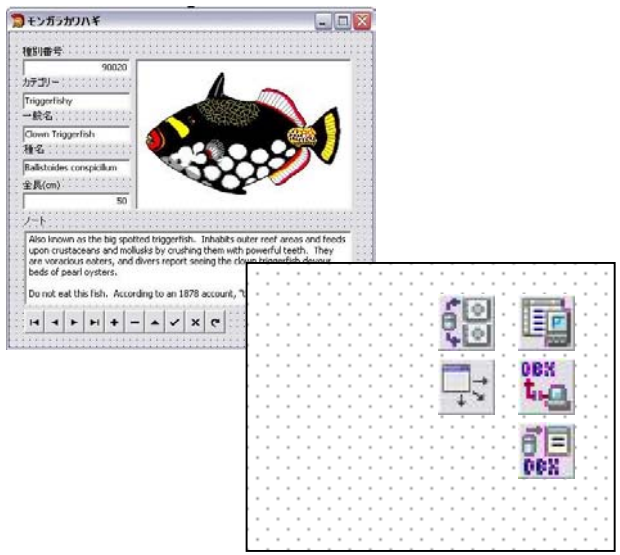
- 利点
  - データロジックの再利用が容易となる
  - メンテナンス性を維持したまま機能を修正、拡張できる
    - 将来的な拡張やフレームワークの変更にも耐えられるように



Borland®

データモジュールを積極的に利用する

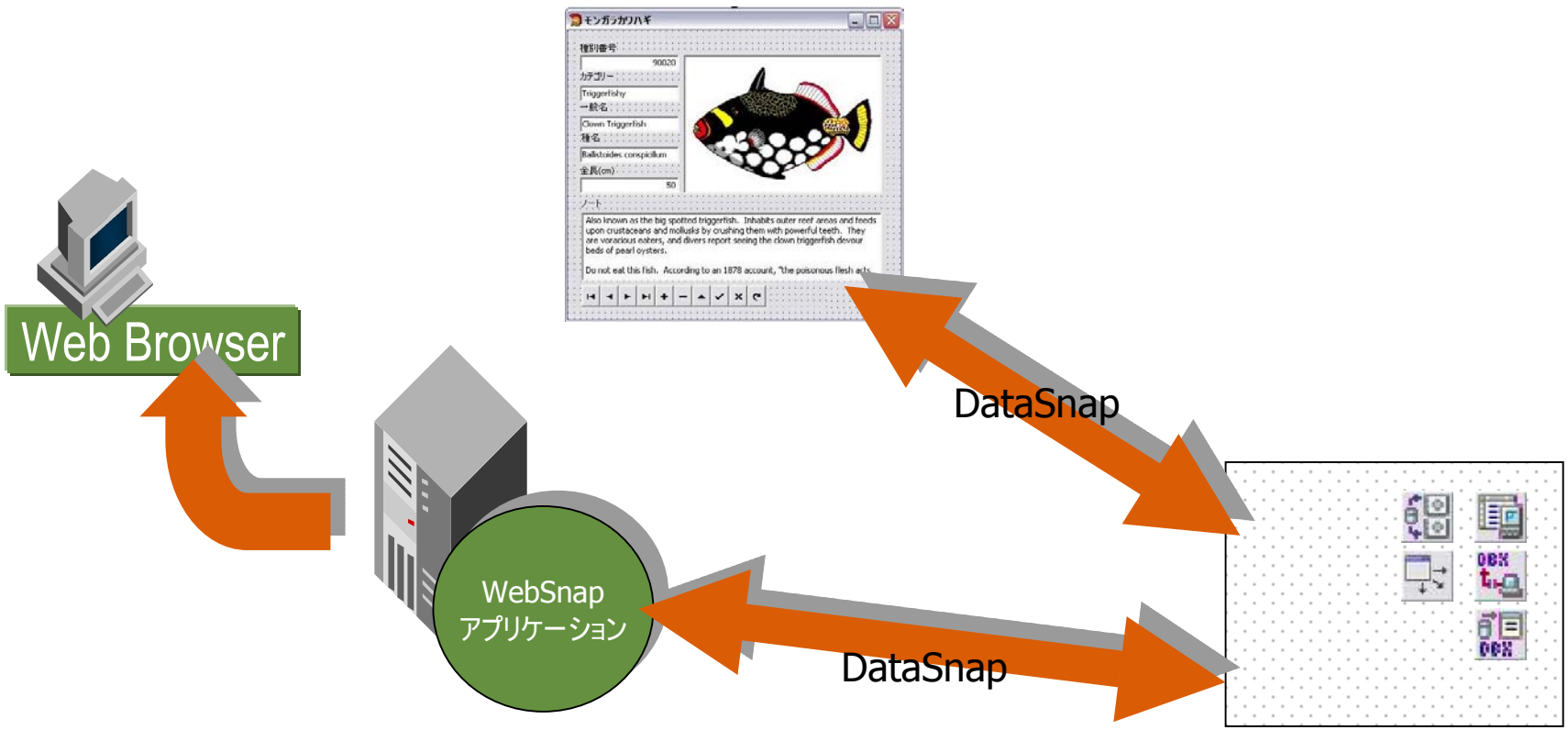
- Snap テクノロジー等と連携して、機能拡張を図りやすくなる



Borland®

データモジュールを積極的に利用する

- Snap テクノロジー等と連携して、機能拡張を図りやすくなる



Borland®

この章のまとめ

- 製品、プラットフォームの機能を正しく利用する
  - OS / ネットワーク / RDBMS / ミドルウェア / クラスライブラリ
  - 正しく利用するための機能の理解は大事である
    - その機能は何のためにある / 何が便利になるの？
      - 機能のメリットとデメリットを把握する
- アプリケーションの設計を考慮した開発
  - 流れに任して開発をしてはいけない
    - 非現実的な実装をなるべく避ける
      - クライアントで処理しきれない多量のデータ取得
      - 不必要なまでの UI 機能はメリットとデメリットを把握した上で実装する
    - ビジュアル開発の利便性のみに注力してはならない
      - 結果、その利便性を開発者自身が享受できなくなる
      - 再利用性やメンテナンス性を考慮する
  - UI と データ操作（ビジネス）ロジックを分離する
    - データモジュールの活用は有効
      - データモジュールは、ただ非 UI コンポーネントを置くだけの場所ではない