Chapter 17
GRASP: Designing Objects with Responsibilities

Understanding responsibilities is key to good object-oriented design.

Martin Flower

Object-Oriented Design

• **OOD is sometimes taught as:**
  - After identifying your requirements and creating a domain model, then **add methods** to the appropriate classes, and **define the messaging** between the objects to fulfill the requirements

• **Such vague advice doesn't help because**
  - Deciding what methods belong where and how objects should interact carries consequences
  - A large set of soft principles, design pattern example, and practice, case study will help to improve your design skills
UML versus Design Principles

- **UML versus Design Principles**
  - The UML is simply a standard visual modeling language, knowing its details doesn't teach you how to think in object
  - The UML is sometimes described as a "design tool" but that's not quite right...
  - The critical design tool for software development is a mind well educated in design principles

Object Design: Example inputs, Activities and Outputs

- **What are inputs to object design?**
  - Some (10-20%) uses cases have been analyzed in detail.
  - Programming experiments have resolved the slow-stopper technical questions.
  - Large-scale logical architecture has been drawn
  - Supplementary Specification
  - Glossary
  - Domain model
  - System Sequence diagram
  - Operation contracts
### Object Design: Example inputs, Activities and Outputs 2

<table>
<thead>
<tr>
<th>The first two-day requirements workshop is finished.</th>
<th>The chief architect and business agree to implement and test some scenarios of Process Sale in the first three-week timeboxed iteration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Three of the twenty use cases that are the most architecturally significant and of high business value have been analyzed in detail, including, of course, the Process Sale use case. UP iterative methods recommends, analyzing only 10%–20% of the requirements in detail before starting to program.</td>
<td>Other artifacts have been started: Supplementary Specification, Glossary, and Domain Model.</td>
</tr>
<tr>
<td>Programming experiments have resolved the show-stopper technical questions, such as whether a Java Swing UI will work on a touch screen.</td>
<td>The chief architect has drawn some ideas for the large-scale logical architecture, using UML package diagrams. This is part of the UP Design Model.</td>
</tr>
</tbody>
</table>

---

### Object Design: Example inputs, Activities and Outputs 3

<table>
<thead>
<tr>
<th>The use case text defines the visible behavior that the software objects must ultimately support objects are designed to implement the use cases. In the UP, this OO design is called the use case realization.</th>
<th>The Supplementary Specification defines the non-functional goals, such as internalization, our objects must satisfy.</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system sequence diagrams identify the system operation messages, which are the starting messages on our interaction diagrams of collaborating objects.</td>
<td>The Glossary clarifies details of parameters or data coming in from the UI layer, data being passed to the database, and detailed item-specific logic or validation requirements, such as the legal formats and validation for product universal product codes.</td>
</tr>
<tr>
<td>The operation contracts may complement the use case text to clarify what the software objects must achieve in a system operation. The post-conditions define detailed achievements.</td>
<td>The Domain Model suggests some names and attributes of software domain objects in the domain layer of the software architecture.</td>
</tr>
</tbody>
</table>
**Object Design: Example inputs, Activities and Outputs**

4

- **What Are Activities of Object Design?**
  - Start immediately coding *(ideally with test-first development)*
  - Start UML modeling for the object design
    - During the drawing (and coding) activity we apply various OO design principles, such as GRASP and the gang-of-four (GoF) design patterns.
    - The overall approach to doing the OO design modeling will be based on the metaphor of responsibility-driven design (RDD), thinking about how to assign responsibilities to collaborating objects.
  - Start with another modeling technique, such as CRC cards

5

- **What’s are the outputs of object design?**
  - UML interaction, class, and package diagrams for the difficult parts of the design that we wished to explore before coding
  - UI sketches and prototypes
  - Database models (with UML data modeling profile notation).
  - Report sketches and prototypes
Responsibility-Driven Design

- We think of software objects as having responsibilities — an abstraction of what they do
- Two types of responsibilities: doing and knowing
  - Doing responsibilities of an object
    - doing something itself, such as creating an object or doing a calculation
    - initiating action in other objects
    - controlling and coordinating activities in other objects
  - Knowing responsibilities of an object
    - knowing about private encapsulated data
    - knowing about related objects
    - knowing about things it can derive or calculate
- Responsibilities are assigned to classes of objects during object design.
  - a Sale is responsible for creating SalesLineItems (doing)
  - a Sale is responsible for knowing its total (knowing).
Responsibility-Driven Design

• **Guideline:** The attributes and associations of domain model often inspires “knowing” responsibilities for software domain objects.
  - If the domain model Sale class has a time attribute, that a software Sale class knows its time (see next page).

• The translation of responsibilities into classes and methods is influenced by the granularity of the responsibility
  - Big responsibilities take hundreds of classes and methods.
    • The responsibility to "provide access to relational databases" may involve two hundred classes and thousands of methods.
  - Little responsibilities might take one method.
    • The responsibility to "create a Sale" may involve only one method in one class.

POS Partial Domain Model

<table>
<thead>
<tr>
<th>Association</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ledger</td>
<td>Sale</td>
</tr>
<tr>
<td>Product Catalog</td>
<td>Store</td>
</tr>
<tr>
<td>Contains</td>
<td>Includes</td>
</tr>
<tr>
<td>ItemID</td>
<td>description</td>
</tr>
<tr>
<td>Describes</td>
<td>*</td>
</tr>
<tr>
<td>Ledgers</td>
<td>Records-accounts for</td>
</tr>
<tr>
<td>Logs</td>
<td>completed</td>
</tr>
<tr>
<td>0..1</td>
<td>1..1</td>
</tr>
<tr>
<td>Sale</td>
<td>Paid-by</td>
</tr>
<tr>
<td>0..1</td>
<td>1</td>
</tr>
<tr>
<td>CashPayment</td>
<td>amountTendered</td>
</tr>
<tr>
<td>Customer</td>
<td>Is-for</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Cashier</td>
<td>Works-on</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0..1</td>
<td>1</td>
</tr>
<tr>
<td>Records-sale-of</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Responsibility-Driven Design

- A responsibility is not the same thing as a method
  - Methods fulfill responsibilities.
- RDD includes the idea of collaboration.
  - Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects.
    - E.g., the Sale class might define a method `getTotal()` to know its total. To fulfill the responsibility, the Sale may collaborate with other objects, such as sending a `getSubtotal` message to each `SalesLineItem` object asking for its subtotal.
- RDD is a general metaphor for thinking about OO software design.
  - Think of software objects as similar to people with responsibilities who collaborate with other people to get work done.

GRASP: A Methodical Approach to Basic OO Design

- GRASP (principles or patterns)
  - General Responsibility Assignment Software Patterns
  - A Learning Aid for OO Design with Responsibilities
    - Apply design reasoning in a methodical, rational, explainable way.
    - To structure and name the principles
    - Understanding and using design principles is based on patterns of assigning responsibilities
Responsibilities/GRASP/UML

• Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities (realized as methods).
  - Sale objects are given a responsibility to create Payments, which is invoked with a makePayment message and handled with a corresponding makePayment method.

```
:Sale
makePayment(cashTended)
```

```
:Payment
create(cashTended)
```

abstract, implies Sale objects have a responsibility to create Payments

when we draw a UML interaction diagram, we are deciding on responsibility assignments

What are Patterns? 1

• Pattern
  - Is a named and well-known problem/solution pair that can be applied in new contexts
    • With advice on how to apply it in novel situations and discussion of its trade-offs, implementations, variations.

<table>
<thead>
<tr>
<th>Pattern Name: Information Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem: What is a basic principle by which to assign responsibilities to objects?</td>
</tr>
<tr>
<td>Solution: Assign a responsibility to the class that has the information needed to fulfill it.</td>
</tr>
</tbody>
</table>

• Naming a pattern, design idea, or principle has the following advantages:
  • It supports chunking and incorporating that concept into our understanding and memory.
  • It facilitates communication
What are Patterns?

- New pattern should be considered an oxymoron if it describes a new idea.
  - The very term “pattern” suggests a long-repeating thing.
  - The point of design patterns is not to express new design ideas. Quite the opposite — great patterns attempt to codify existing tried-and-true knowledge, idioms, and principles.

- The Gang-of-Four Design Patterns.
  - 23 patterns (Strategy, Adaptor, …)

- GRASP defines 9 basic OO design principles (or basic building blocks in design).
  - Doesn’t GRASP describe principles rather than patterns?
    - One person’s pattern is another person’s primitive building block.
    - Using the pattern style as an excellent learning aid for naming, presenting, and remembering basic, classic design ideas.

Nine GRASP patterns

- There are 9 GRASP patterns
  - Creator
  - Controller
  - Information Expert
  - High Cohesion
  - Low Coupling
  - Pure Fabrication
  - Indirection
  - Polymorphism
  - Protected variations
Creator 1

• Problem
  - Who should be responsible for creating a new instance of some class?
  - Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.

• Solution (advices)
  - Assign class B the responsibility to create an instance of class A if one of these is true
    • B “contains” or compositely aggregates A.
    • B records A.
    • B closely uses A.
    • B has the initializing data for A that will be passed to A when it is created. (Thus B is an Expert with respect to creating A.)
  - Prefer a class B which aggregates or contains class A

Creator 2

• Example
  - NextGen POS: who should be responsible for creating a SalesLineItem instance?
    • By Creator, look for a class that aggregates, contains SalesLineItem instances.
**Creator 3**

- A Sale contains (aggregates) many SalesLineItem objects
  - Creator pattern suggests that Sale is a good candidate to have the responsibility of creating SalesLineItem instances.
  - This leads to the design of object interactions shown below
  - This assignment of responsibilities requires that a makeLineItem method be defined in Sale

```
Register
  makeLineItem(quantity)

SalesLineItem
  Create(quantity)
```

**Creator 4**

- **Discussion**
  - The concept of composition (Composite aggregates Part, Container contains Content, and Recorder records)
  - Expert pattern: initializing data is passed in during creation via some kind of initialization method, such as a java constructor that has parameters.
    - Assume that a payment instance, when created, needs to be initialized with the sale total. Since sale knows the total, sale is a candidate creator of the payment.
- **Contraindications**
  - If creation requires significant complexity, such as using recycled instances for performance, conditionally creating an instance from one of a family of similar classes based upon some external property value.
    - Delegate creation to a helper class called a Concrete Factory or an Abstract Factory rather than use Creator.
**Creator**

- **Benefits**
  - Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse.
  - Coupling is not increased because the created class is visible to the creator class, due to the existing associations.

- **Related Patterns or Principles**
  - Low Coupling
  - Concrete Factory and Abstract Factory
  - Whole-Part describes a pattern to define aggregate objects that support encapsulation of components.

---

**Information Expert**

- **Problem**
  - What is a general principle of assigning responsibilities to objects?
  - A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled.
  - If we've chosen well → easier to understand, maintain, and extend → more opportunity to reuse components

- **Solution**
  - Assign a responsibility to the information expert — the class that has the information necessary to fulfill the responsibility.
**Information Expert 2**

- **Example**
  - NextGEN POS: some class needs to know the grand total of all the SalesLineItem instances of a sale.
  - Who should be responsible for knowing that?
    - If there are relevant classes in the Design Model, look there first.
    - Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes.
  - **Information Expert of the Domain Model**: look for class that has the information needed to determine the total - Sale.
    - Give the responsibility of knowing its total - method getTotal.
    - This approach supports low representational gap between software design of objects and the concepts of real domain.

```
Sale
  time
  1 Contains 1..* Described-by
  SalesLineItem
    quantity
  ProductDescription
    description
    price
    itemID
```

**Information Expert 3**

- To create the interaction diagrams in order to assign responsibilities to objects.
- Who should know the line item subtotal?
  - The SalesLineItem knows its quantity and its associated ProductDescription; therefore, by Expert, SalesLineItem should determine the subtotal.
  - For interaction diagram, the Sale should send getSubtotal messages to each of the SalesLineItems and sum the results

```
Sale
  time
  st=getTotal
SalesLineItem
  st=getSubtotal
```

this notation will imply we are interacting over all elements of a collection

New method

```
SaleLineItem
  quantity
  getSubtotal()()
```

Sale
  time
  ...
Information Expert 4

- To fulfill the responsibility of knowing its subtotal, a SalesLineItem has to know the product price.
- The ProductDescription is an information expert on answering its price.
- SalesLineItem sends a message asking for the product price

![](image)

Information Expert 5

- To fulfill the responsibility of knowing and answering the sale’s total.
  - Assign three responsibilities to three design classes of objects in the interaction diagram.
  - Summarize the methods in the method section of a class diagram.

<table>
<thead>
<tr>
<th>Design Class</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sale</td>
<td>knows sale total</td>
</tr>
<tr>
<td>SalesLineItem</td>
<td>knows line item subtotal</td>
</tr>
<tr>
<td>ProductDescription</td>
<td>knows product price</td>
</tr>
</tbody>
</table>
Discussion
- The fulfillment of a responsibility often requires information crossing different classes of objects.
  - Many "partial" information experts will collaborate in the task.
  - The sales total problem required the collaboration of three classes of objects. they need to interact via messages to share the work.
- "Do It Myself" strategy.
  - In object-oriented software, all software objects are alive, and they can take on responsibilities and do things related to the information they know.
- We commonly give responsibility to individuals who have the information necessary to fulfill a task (real-world analogy)
  - In a business, the chief financial officer should be responsible for creating a profit-and-loss statement.
    - Software objects collaborate because the information is spread around, so it is with people. The company's chief financial officer may ask accountants to generate reports on credits and debits.

Contraindications
- In some situations, a solution suggested by expert is undesirable, usually because of coupling, cohesion, and duplication problems.
- E.g., Who should be responsible for saving a sale in a database? Sale class.
  - The sale class must contain database handling, SQL and JDBC.
  - Lower cohesion: the class no longer focuses on just the pure application logic of "being a sale."
  - Increase coupling: the class must be coupled to the technical database/JDBC services of another subsystem, rather than being coupled to other objects in the domain layer.
  - Similar database logic would be duplicated in many persistent classes.
- The above violate basic architectural principle
  - Design for a separation of major system concerns.
  - Keep application logic in one place (domain software objects).
  - Keep database logic in another place
Information Expert

• Benefits
  - Information encapsulation is maintained since objects use their own information to fulfill tasks. This usually supports low coupling.
  - Behavior is distributed across the classes that have the required information, encouraging more cohesive “lightweight” class definitions that are easier to understand and maintain.

• Related Patterns or Principles
  - Low Coupling
  - High Cohesion

• Also Known As
  - “Place responsibilities with data,” “That which knows, does”
  - “Do It Myself,” “Put Services with the Attributes They Work On.”

---

class Foo {
public:
  void SetValue(int);
  int GetValue();
private:
  int value;
};

Some other class ...

Foo *foo = new Foo();
...
int v = foo->GetValue();
if (v >= 100)
  foo->SetValue(v * 0.8);
else
  foo->SetValue(v * 0.9);
...

What’s wrong?

Violates information expert

class Foo {
    public:
        void SetValue(int);
        int GetValue();
        void Discount() {
            if (value > 100)
                value *= 0.8;
            else
                value *= 0.9
        }
    private:
        int value;
};

class Hero {
    ... 
    void Move() {
        if (map->GetCell(x, y) == 1)
            ... 
        else 
            ... 
        }
    Map *map;
    int x, y;
};

Better!!!

What's wrong?
class Map {
    public:
    ... int GetCell(int x, int y)
        { return cell[x][y]; }
    bool isWalkable(int x, int y)
        { return cell[x][y] == 1; }
    private:
    int cell[100][100]
};

class Hero {
    ... void Move()
        { if (map->isWalkable(x,y))
            ... else
            ... }
    Map *map;
    int x, y;
}

Information Expert

Low Coupling

• Problem
  - How to support low dependency, low change impact, and increased reuse?

• Solution
  - Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives. (next page)
  - Coupling

  – A measure of how strongly one element is connected to, has knowledge of, or relies on other elements.
  – An element with low coupling is not dependent on too many other classes, subsystems, systems.
  – High coupling problems:
    - Forced local changes because of changes in related classes.
    - Harder to understand in isolation.
    - Harder to reuse because its use requires the additional presence of the classes on which it is dependent.
Low Coupling

High coupling

Low coupling

Example

NextGen POS

To create a Payment instance and associate it with the Sale. What class should be responsible for this?

Register   Sale

Which design, based on assignment of responsibilities, supports Low Coupling?

In both cases we assume the Sale must eventually be coupled to knowledge of a Payment.
Low Coupling

- Design 1
  - Since a Register "records" a Payment in the real-world domain, the Creator pattern suggests Register as a candidate for creating the Payment.
  - The Register instance could then send an addPayment message to the Sale, passing along the new Payment as a parameter.
  - **Adds coupling of** Register to Payment

```
makePayment() :Register 1:create()  →  p:Payment
               ↓  2:addPayment(p) → :Sale
```

- Design 2
  - **the Sale does the creation of a Payment, does not increase the coupling.**
  - Purely from the point of view of coupling, prefer Design 2 because it maintains overall lower coupling.
  - In practice, the level of coupling alone can’t be considered in isolation from other principles such as Expert and High Cohesion.

```
makePayment()  :Register 1:makePayment()   →  :Sale
               ↓     1.1:create()  ↓  
                    ↓  p:Payment
```
Low Coupling

- **Discussion**
  - Low Coupling is a principle to keep in mind during all design decisions.
  - It is an evaluative principle that you apply while evaluating all design decisions.
  - In object-oriented languages such as C++, Java, and C#, common forms of coupling from TypeX to TypeY include:
    - TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
    - A TypeX object calls on services of a TypeY object.
    - TypeX has a method that references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
    - TypeX is a direct or indirect subclass of TypeY.
    - TypeY is an interface, and TypeX implements that interface.

- **Contraindications**
  - High coupling to stable elements and to pervasive elements is seldom a problem.
    - J2EE application can safely couple itself to the Java libraries (Java.util...), because they are stable and widespread.
Low Coupling

- **Pick Your Battles**
  - It is not high coupling per se that is the problem; it is high coupling to elements that are unstable in some dimension, such as their interface, implementation, or mere presence.
  - Focus on the points of realistic high instability or evolution.
    - The NextGen project, the different third-party tax calculators (with unique interfaces) need to be connected to the system. Designing for low coupling at this variation point is practical.

- **Benefits**
  - Not affected by changes in other components
  - Simple to understand in isolation
  - Convenient to reuse

- **Related Patterns**
  - Protected Variation

---

class Map {
public:
    ... 
    int GetCell(int x, int y)
    {
        return cell[x][y];
    }
    bool isWalkable(int x, int y)
    {
        return cell[x][y] == 1;
    }
private:
    int cell[100][100]
};

class Hero {
    ...
    void Move()
    {
        if (map->isWalkable(x,y))
            ...
            One-way dependency
        else
            ...
    }
    Map *map;
    int x, y;
};
Low Coupling

```cpp
class Map {
    public:
        ...
        bool isWalkable(Hero *h) {
            return cell[h->GetX()][h->GetY()] == 1;
        }
    private:
        int cell[100][100]
};
```

```cpp
class Hero {
    ...
    void Move() {
        if (map->IsWalkable(this))
            ...
        else
            ...
    }
    Map *map;
    int x, y;
};
```

Controller

1. **Problem**
   - What first object beyond the UI layer receives and coordinates ("controls") a system operation?
     - A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.
     - System operations were first explored during the analysis of SSD (next page). These are the major input events upon our system. e.g.,
       - When a cashier using a POS terminal presses the "end sale" button (next page), he is generating a system event indicating "the sale has ended."
       - When a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."
Controller 1a

Simple cash-only Process Sale scenario:
1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.

Controller 2

• Solution
  - Assign the responsibility to a class that
    • Represents the overall “system,” a “root object,” a device that the software is running within, or a major subsystem. These are all variations of a facade controller.
    • Represents a use case scenario within which the system event occurs, often named <UseCaseName> Handler, <UseCaseName> Coordinator, or <UseCaseName> Session (use case or session controller).
      - Use the same controller class for all system events in the same use case scenario.
      - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).
    • “Window,” “view,” and “document” classes are not on this list. They typically receive these events and delegate them to a controller.
Controller 3

- Example
  - NextGen POS
    - Some system operations shown.
    - This model shows the system itself as a class (which is legal and sometimes useful when modeling).

<table>
<thead>
<tr>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>endSale()</td>
</tr>
<tr>
<td>enterItem()</td>
</tr>
<tr>
<td>makeNewSale()</td>
</tr>
<tr>
<td>makePayment()</td>
</tr>
<tr>
<td>. . .</td>
</tr>
</tbody>
</table>

Controller 4

- During analysis, system operations may be assigned to the class System.
- During design, a controller class is assigned the responsibility for system operations.

Which class of object should be responsible for receiving this System event message? It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects. The controller is a kind of façade onto the domain layer from the interface layer.
Controller

- Who should be the controller for system events such as `enterItem` and `endSale`?
  - By the Controller pattern
  - Represents the overall "system," "root object," device, or subsystem. (Register, POSSystem)
  - A receiver or handler of all system events of a use case scenario. (ProcessSaleHandler, ProcessSaleSession)
  - The domain of POS, a Register (POS Terminal) is a specialized device with software running in it.
  - The interaction diagrams:

```
enterItem(id, quantity) -> :Register

enterItem(id, quantity) -> :ProcessSaleHandler
```

Controller

- The choice of which of these classes is the most appropriate controller is influenced by other factors
  - During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such as Register
  - System operations discovered during system behavior analysis
  - Allocation of system operations during design using one facade controller
  - Allocation of system operations during design using several use case controllers
**Controller 7**

- **Discussion**
  - This is a delegation pattern.
    - The UI layer shouldn’t contain application logic.
    - UI layer objects must delegate work requests to domain layer.
    - In the domain layer, the Controller pattern summarizes choices, make for the domain object delegate that receives the work requests.
    - The controller is a kind of facade into the domain layer from the UI layer.
  - You will often want to use the same controller class for all the system events of one use case so that the controller can maintain information about the state of the use case.
    - To identify out-of-sequence system events (e.g., a makePayment operation before an endSale operation).
    - Different controllers may be used for different use cases.

---

**Controller 8**

- **Discussion (continued)**
  - A common defect in the design of controllers results from over-assignment of responsibility.
    - Then, the controller suffers from bad/low cohesion, violating the principle of High Cohesion.
  - **Guideline:** a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. *It does not do much work itself.*
  - **UP Boundary, Control, and entity classes**
    - Boundary objects are abstractions of the interfaces
    - Entity objects are the application-independent/persistent (typically persistent) domain objects
    - Control objects are use case handlers as described in this Controller pattern.
Controller 9

- Discussion (continued)
  - The first category of controller is a facade controller representing the overall system, device, or a subsystem
    - The idea is to choose some class name that suggests a cover, or facade, over the other layers of the application and that provides the main point of service calls from the UI layer down to other layers
    - The facade could be an abstraction of the overall physical unit, such as a Register; a class representing the entire software system, such as POSSystem.
    - Facade controllers are suitable when there are not “too many” system events, or when the UI cannot redirect system event messages to alternating controllers, such as in a message-processing system.
  - If you choose a use case controller, then you will have a different controller for each use case.
    - Note that this kind of controller is not a domain object; it is an artificial construct to support the system (a Pure Fabrication in terms of the GRASP patterns).

Controller 10

- Discussion (continued)
  - Web UIs and Server-Side Application of Controller
    - A delegation approach can be used in ASP.NET WebForms:
      - The “code behind” file that contains event handlers for Web browser button clicks will obtain a reference to a domain controller object (e.g., a Register object in the POS), and then delegate the request for work.
    - Server-side Web UI frameworks (such as Struts) embody the concept of the Web-MVC (Model-View-Controller) pattern
      - In Web-MVC, the “controller” is part of the UI layer and controls the UI interaction and page flow.
      - The GRASP controller is part of the domain layer and controls the handling of the work request, essentially unaware of what UI technology is being used (e.g., a Web UI, a Swing UI).
Controller 11

- Discussion (continued)
  - Web UIs and Server-Side Application of Controller
    - Java server-side design is delegation from the Web UI layer to an EJB Session object. The EJB Session object may itself delegate farther on to the domain layer of objects, and to apply the Controller pattern to choose a suitable receiver in the pure domain layer.
      - Variant #2 of the Controller pattern — an object representing a user session or use case scenario — covers this case
    - A rich-client UI (e.g., Swing)
      - forwards the request to the local client-side controller, and the controller forwards all or part of the request handling to remote services. This design lowers the coupling of the UI to remote services and makes it easier, for example, to provide the services either locally or remotely, through the indirection of the client-side controller.

Controller 12

- Benefits
  - Increased potential for reuse and pluggable interfaces
    - Application logic is not handled/bound in the interface layer, it can be replaced with a different interface.
    - Delegating a system operation responsibility to a controller supports the reuse of the logic in future applications.
  - Opportunity to reason about the state of the use case.
    - To ensure that system operations occur in a legal sequence, or we want to be able to reason about the current state of activity and operations within the use case.
    - E.g., to guarantee that the makePayment operation cannot occur until the endSale operation has occurred. To capture this state information somewhere; the controller is one reasonable choice.
**Controller 13**

- **Implementation**
  - **Bloated Controllers:** has low cohesion unfocused and handling too many areas of responsibility
    - There is only a single controller class receiving all system events in the system, and there are many of them.
    - The controller performs many of the tasks to fulfill the system event, without delegating the work.
    - A controller has many attributes, and maintains significant information about the system, which should have been distributed to other objects, or it duplicates information found elsewhere.
  - The cures for a bloated controller
    - Add more controllers — a system does not have to need only one. Instead of facade controllers, employ use case controllers.
      - E.g., an application with many system events - an airline reservation system - contain use case controllers: MakeReservationHandler, ManageSchedulesHandler, ManageFaresHandler
    - Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects.

**Controller 14**

- **UI Layer Does Not Handle System Events**
  - Assume the NextGen application has a window that displays sale information and captures cashier operations.
  - Using the Controller pattern illustrates an acceptable relationship between the JFrame and the controller and other objects in a portion of the POS system.
  - SaleJFrame class (part of the UI layer) delegates the enterItem request to the Register object (next page).
  - If a UI layer object (SaleJFrame) handles a system operation that represents part of a business process (next page)
    - Business process logic would be contained in an interface object;
    - The opportunity for reuse of the business logic then diminishes because of its coupling to a particular interface and application.
Controller

Good

Cashier

actionPerformed(actionEvent)

UI Layer

SaleJFrame

system operation message

Domain Layer

Register

controller

Controller

Bad

Cashier

actionPerformed(actionEvent)

UI Layer

SaleJFrame

It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain process. Business logic is embedded in the Presentation layer, which is not useful.

Domain Layer

1:makeLineItem(itemID, qty)

Sale

SaleJFrame should not sending message
Controller

• Related Patterns
  - Command: In a message-handling system, each message may be represented and handled by a separate Command object.
  - Facade: A facade controller is a kind of Facade [GHJV95].
  - Layers: This is a POSA pattern. Placing domain logic in the domain layer rather than the presentation layer is part of the Layers pattern.
  - Pure Fabrication: This GRASP pattern is an arbitrary creation of the designer, not a software class whose name is inspired by the Domain Model. A use case controller is a kind of Pure Fabrication.

High Cohesion

• Problem
  - How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - Cohesion: a measure of how strongly related and focused the responsibilities of a class or subsystem are.
  - Low cohesion class
    • Does many unrelated things or too much work.
    • Represent a very “large grain” of abstraction or have taken on responsibilities that should have been delegated to other objects
    • Hard to comprehend, reuse, maintain
    • Delicate (constantly affected by change)

• Solution
  - Assign a responsibility so that cohesion remains high.
  - Use this to evaluate alternatives.
High Cohesion 2

• Example
  - POS: Low Coupling pattern for High Cohesion.
    - To create a (cash) Payment instance and associate it with the Sale. What class should be responsible for this?

```
// Design 1
- Since Register records a Payment in the real-world domain, the Creator pattern suggests Register for creating the Payment.
- The Register instance could send an addPayment message to the Sale, passing along the new Payment as a parameter
- To place the responsibility for making a payment in the Register.

• To continuously make the Register class responsible for doing most of the work related to more system operations
  - It will become increasingly with tasks and become incohesive.
```

[Diagram of class relationships]
High Cohesion 4

- The second design delegates the payment creation responsibility to the Sale supports higher cohesion in the Register.
  - The second design is desirable, since it supports both high cohesion and low coupling.
- In practice, the level of cohesion alone can’t be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.

![Diagram]

High Cohesion 5

- Discussion
  - **Very low cohesion:** A class is responsible for many things in very different functional areas.
  - **RDB-RPC-Interface class is completely responsible for interacting with relational databases and for handling remote procedure calls.**
  - **The responsibilities should be split into a family of classes related to RDB access and a family related to RPC support.**
  - **Low cohesion:** A class has responsibility for a complex task in one functional area.
    - **RDBInterface class is completely responsible for interacting with relational databases.**
    - **The methods of the class are all related, but hundreds or thousands of methods.**
    - **The class should split into a family of lightweight classes sharing the work to provide RDB access.**
High Cohesion 6

Discussion
- **High cohesion**: A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
  - **RDBInterface class** is only partially responsible for interacting with relational databases.
  - It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.
- **Moderate cohesion**: A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.
  - **Company class** is completely responsible for (1) knowing its employees and (2) knowing its financial information.
  - These two areas are not strongly related to each other, although both are logically related to the concept of a company.
  - The total number of public methods is small.

High Cohesion 7

Discussion
- **High cohesion class has a relatively small number of methods, with highly related functionality, and does not do too much work.** It collaborates with other objects to share the effort if the task is large.
- **This principle is an evaluative principle evaluating all design decisions.**
- **Real world**
  - if a person takes on too many unrelated responsibilities, then the person is not effective.
  - Managers have not learned how to delegate, they are ready to become “unglued.”
High Cohesion

- Modular Design Principle
  - A system has been decomposed into a set of cohesive and loosely coupled modules.
  - We promote a modular design by creating methods and classes with high cohesion

- Contraindications
  - Case 1
    - Only one or two SQL experts know how to best define and maintain SQL.
    - Few OO programmers may have strong SQL skills.
    - Suppose the SQL expert is not even a comfortable OO programmer.
    - The software architect may decide to group all the SQL statements into one class, RDBOperations, so that it is easy for the SQL expert to work on the SQL in one location.

High Cohesion

- Contraindications
  - Case 2: Lower cohesion is with distributed server objects.
    - Because of overhead and performance implications associated with remote objects and remote communication, it is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations.
    - This approach is also related to the pattern called Coarse-Grained Remote Interface
      - For example, instead of a remote object with three fine-grained operations setName, setSalary, and setHireDate, there is one remote operation, setData, which receives a set of data. This results in fewer remote calls and better performance.
High Cohesion

• Benefits
  - Clarity and ease of comprehension of the design is increased.
  - Maintenance and enhancements are simplified.
  - Low coupling is often supported.
  - Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.

Cohesion and Coupling; Yin and Yang

• Bad cohesion usually begets bad coupling, and vice versa.
  - Cohesion and coupling are the yin and yang of software engineering
  - For example, consider a GUI widget class that represents and paints a widget, saves data to a database, and invokes remote object services. Not only is it profoundly incohesive, but it is coupled to many disparate elements.
Chapter 18
Object Design
Examples with GRASP

What is a Use Case Realization? 1

- **Key Point of Object Design**
  - The assignment of responsibilities and design of collaborations are very important and creative steps during design, both while diagramming and while coding.
- **A use-case realization describes how a particular use case is realized within the Design Model, in terms of collaborating objects (RUP)**
  - More precisely, a designer can describe the design of one or more scenarios of a use case; each of these is called a use case realization.
  - Use case realization is a UP term used to remind us of the connection between the requirements expressed as use cases and the object design that satisfies the requirements.
What is a Use Case Realization?

- Some relevant artifact-influence points include the following
  - The use case suggests the system operations that are shown in SSDs
  - The system operations become the starting messages entering the Controllers for domain layer interaction diagrams
    - A key point often missed by those new to OOA/D modeling
  - Domain layer interaction diagrams illustrate how objects interact to fulfill the required tasks the use case realization
Artifact Comments

- SSDs, System Operations, Interaction Diagrams, and Use Case Realizations
  
  - In the current NextGen POS iteration we are considering scenarios and system operations identified on the SSDs of the Process Sale use case
    
    - makeNewSale
    - enterItem
    - endSale
    - makePayment
  
  - If we use communication diagrams to illustrate the use case realizations
    
    - we will draw a different communication diagram to show the handling of each system operation message.
    - Of course, the same is true for sequence diagrams

Communication Diagrams and System Operation Handling

Window objects or GUI widget objects or Web control objects

makeNewSale → :Register

enterItem → :Register

endSale → :Register

makePayment → :Register

UI LAYER DOMIAN LAYER
Sequence Diagrams and System Operation Handling

![Sequence Diagrams and System Operation Handling](image)

Artifact Comments 1

- **Use Cases and Use Case Realizations**
  - Use cases are a prime input to use case realizations.
    - The use case text and related requirements expressed in the Supplementary Specifications, Glossary, UI prototypes, report prototypes, and so forth, all inform developers what needs to be built.
  - **Involve the Customer Frequently**
    - Documents are the critical requirements input to doing software design and development, however, it is hard to beat the ongoing participation of customers in evaluating demos, discussing requirements and tests, prioritizing, and so forth
Artifact Comments 2

- **Operation Contracts** and Use Case Realizations
  - Use case realizations could be designed directly from the use case text or from one’s domain knowledge. For some complex system operations, contracts may have been written that add more analysis detail.
  - In conjunction with contemplating the use case text, for each contract, we work through the postcondition state changes and design message interactions to satisfy the requirements

**Contract CO2: enterItem**

Operation: enterItem(itemID: ItemID, quantity: integer)
Cross References: Use Cases: Process Sale
Preconditions: There is a sale underway.
Postconditions:
  - A SalesLineItem instance sli was created (instance creation).

...
Artifact Comments

- The Domain Model and Use Case Realizations
  - In the interaction diagrams, the Domain Model inspires some of the software objects
  - The existing Domain Model — as with all analysis artifacts — won't be perfect
    - You should expect errors and omissions. You will discover new concepts that were previously missed, ignore concepts that were previously identified, and do likewise with associations and attributes
  - Must you limit the design classes in the Design Model to classes with names inspired from the Domain Model?
    - Not at all. It's normal to discover new conceptual classes during design work that were missed during earlier domain analysis and to make up software classes whose names and purpose are completely unrelated to the Domain Model

Use Case Realizations for the NextGen Iteration

- Initialization and the 'Start Up' Use Case
  - The Start Up use case realization is the design context in which to consider creating most of the 'root' or long-lived objects
  - Guideline
    - When coding, program at least some Start Up initialization first. But during OO design modeling, consider the Start Up initialization design last, after you have discovered what really needs to be created and initialized. Then, design the initialization to support the needs of other use case realizations
Use Case Realizations for the NextGen Iteration 2

• How to Design makeNewSale?
  - The makeNewSale system operation occurs when a cashier initiates a request to start a new sale, after a customer has arrived with things to buy

**Contract C01: makeNewSale**

Operation: makeNewSale()  
Cross References: Use Cases: Process Sale  
Preconditions: none  
Postconditions:
  A Sale instance s was created (instance creation).  
s was associated with a Register (association formed).  
Attributes of s were initialized.

Use Case Realizations for the NextGen Iteration 3

• Choosing the Controller Class (SSD and UCD next page)
• Our first design choice involves choosing the controller for the system operation message enterItem. By the Controller pattern, here are some choices:
  - Represents the overall "system," "root object," a specialized device, or a major subsystem.
    • Store — a kind of root object because we think of most of the other domain objects as "within" the Store.
    • Register — a specialized device that the software runs on; also called a POSTerminal.
    • POSSystem — a name suggesting the overall system
  - Represents a receiver or handler of all system events of a use case scenario.
    • ProcessSaleHandler — constructed from the pattern <use-case-name> "Handler" or "Session" ProcessSaleSession
Use Case Realizations for the NextGen Iteration 4

- Choosing a device-object facade controller like Register is satisfactory if there are only a few system operations and if the facade controller is not taking on too many responsibilities (in other words, if it is not becoming incohesive).
- Choosing a use case controller is suitable when we have many system operations and we wish to distribute responsibilities in order to keep each controller class lightweight and focused (in other words, cohesive).
- In this case, Register suffices since there are only a few system operations
Use Case Realizations for the NextGen Iteration

• Creating a New Sale
  - GRASP Creator pattern suggests assigning the responsibility for creation to a class that aggregates, contains, or records the object to be created
    • Analyzing the Domain Model (next page) reveals that a Register may be thought of as recording a Sale
  - When the Sale is created, it must create an empty collection (such as a Java List) to record all the future SalesLineItem instances that will be added.
    • This collection will be contained within and maintained by the Sale instance, which implies by Creator that the Sale is a good candidate for creating the collection

POS Partial Domain Model
**Use Case Realizations for the NextGen Iteration 6**

- **Register creates a Sale by Creator**
- **by Creator, Sale creates an empty collection (such as a List) which will eventually hold SalesLineItem instances**

**How to Design enterItem?**
- The `enterItem` system operation occurs when a cashier enters the `itemId` and (optionally) the quantity of something to be purchased.

**Contract CO2: enterItem**
- Operation: `enterItem(itemId: ItemID, quantity: integer)`
- Cross References: Use Cases: Process Sale
- Preconditions: There is a sale underway.
- Postconditions:
  - A SalesLineItem instance `sli` was created (instance creation).
  - `sli` was associated with the current Sale (association formed).
  - `sli.quantity` became `quantity` (attribute modification).
  - `sli` was associated with a ProductDescription, based on `itemId match` (association formed).
Use Case Realizations for the NextGen Iteration 7

• Choosing the Controller Class
  - Based on the Controller pattern, as for makeNewSale, we will continue to use Register as a controller

• Display Item Description and Price?
  - Because of a principle of Model-View Separation, it is not the responsibility of non-GUI objects (such as a Register or Sale) to get involved in output tasks.
  - Ignore the design at this time

• Creating a New SalesLineItem
  - Analysis of the Domain Model reveals that a Sale contains SalesLineItem objects
    - By Creator, a software Sale is an appropriate candidate to create a SalesLineItem
    - The postconditions indicate that the new SalesLineItem needs a quantity when created; therefore, the Register must pass it along to the Sale, which must pass it along as a parameter in the create message.

Use Case Realizations for the NextGen Iteration 8

• Finding a ProductDescription
  - The SalesLineItem needs to be associated with the ProductDescription that matches the incoming itemID. This implies that we must retrieve a Product-Description, based on an itemID match
    - By Information Expert ProductCatalog is a good candidate for this lookup responsibility since it knows all the ProductDescription objects

• Visibility to a ProductCatalog
  - Who should send the getProductDescription message to the ProductCatalog to ask for a ProductDescription?
    - Visibility is the ability of one object to "see" or have a reference to another object
      - For an object to send a message to another object, it must have visibility to it
      - The Register has a permanent connection or reference to the ProductCatalog, it has visibility to it
Question: The coupling between Register and ProductCatalog appears to be unnecessary. Why not eliminate the coupling by creating an association from SalesLineItem directly?

Answer: (1) the coupling is useful for some other operations (2) Register has a longer life cycle; SalesLineItem is dynamically created (can not create an association by itself)
Use Case Realizations for the NextGen Iteration 11

• How to Design endSale?
  - The endSale system operation occurs when a cashier presses a button indicating the end of entering line items into a sale

  Contract CO3: endSale
  Operation: endSale()
  Cross References: Use Cases: Process Sale
  Preconditions: There is a sale underway.
  Postconditions:
  Sale.isComplete became true (attribute modification).

• Choosing the Controller Class
  - Based on the Controller GRASP pattern, as for enterItem, we will continue to use Register as a controller

Use Case Realizations for the NextGen Iteration 12

• Setting the Sale.isComplete Attribute
  - As always, Expert should be the first pattern considered unless the problem is a controller or creation problem (which it is not).
  - By Expert, it should be the Sale itself, since it owns and maintains the isComplete attribute. Thus, the Register will send a becomeComplete message to the Sale to set it to true
Use Case Realizations for the NextGen Iteration

• Calculating the Sale Total
  
  - Consider this fragment of the Process Sale use case
    
    Main Success Scenario:
    3. Customer arrives ...
    4. Cashier tells System to create a new sale.
    5. Cashier enters item identifier.
    6. System records sale line item and ...
      - Cashier repeats steps 3-4 until indicates done.
    7. System presents total with taxes calculated
      
      - In step 7, a total is presented (or displayed). Because of the Model-View Separation principle, we should not concern ourselves with the design of how the sale total will be displayed, but we must ensure that the total is known.

• Who should be responsible for knowing the sale total?

• Summarize the information required:
  - The sale total is the sum of the subtotals of all the sales line-items.
  - sales line-item subtotal := line-item quantity *product description price

• List the information required to fulfill this responsibility and the classes that know this information

<table>
<thead>
<tr>
<th>Information Required for Sale Total</th>
<th>Information Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProductDescription.price</td>
<td>ProductDescription</td>
</tr>
<tr>
<td>SalesLineItem.quantity</td>
<td>SalesLineItem</td>
</tr>
<tr>
<td>all the SalesLineItems in the current Sale</td>
<td>Sale</td>
</tr>
</tbody>
</table>
Use Case Realizations for the NextGen Iteration

- The Sale.getTotal Design
  - The first message in this diagram is `getTotal`, but observe that the `getTotal` message is not a system operation message
  - Not all interaction diagrams start with a system operation message; they can start with any message for which the designer wishes to show interactions

```
by Expert

:Sale
tot = getTotal

1 * [i = 1..n]: s = getSubtotal

:ProductDescription

by Expert

1.1: pr = getPrice
```

- Since arithmetic is not (usually) illustrated via messages, we can illustrate the details of the calculations by attaching algorithms or constraints to the diagram that defines the calculations
- Who will send the `getTotal` message to the Sale? Most likely, it will be an object in the UI layer, such as a Java JFrame

```
public void getTotal()
{
    int tot = 0;
    for each SalesLineItem s:
        tot = tot + s.getSubtotal();
    return tot;
}
```

- Since arithmetic is not (usually) illustrated via messages, we can illustrate the details of the calculations by attaching algorithms or constraints to the diagram that defines the calculations
- Who will send the `getTotal` message to the Sale? Most likely, it will be an object in the UI layer, such as a Java JFrame
Use Case Realizations for the NextGen Iteration

• How to Design makePayment?
  - The makePayment system operation occurs when a cashier enters the amount of cash tendered for payment

  Contract CO4: makePayment
  Operation: makePayment( amount: Money )
  Preconditions: There is a sale underway.
  Postconditions:
  A Payment instance p was created (instance creation).
  p.amountTendered became amount (attribute modification).
  p was associated with the current Sale (association formed).
  The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales)

Use Case Realizations for the NextGen Iteration

• Creating the Payment
  - By creator, we have two candidates: Register and Sale
  - By Expert, Register has the initializing data for Payment (the amount tendered by customer)
  - Guideline: When there are alternative design choices, take a closer look at the cohesion and coupling implications of the alternatives, and possibly at the future evolution pressures on the alternatives. Choose an alternative with good cohesion, coupling, and stability in the presence of likely future changes
Use Case Realizations for the NextGen Iteration 19

- **Design Choice**
  - If we choose the Sale to create the Payment, the work (or responsibilities) of the Register is lighter — leading to a simpler Register definition.
  - Also, the Register does not need to know about the existence of a Payment instance because it can be recorded indirectly via the Sale — leading to lower coupling in the Register.

```
makePayment(cashTendered)  
```

What happens if we choose the register to create the Payment?

Use Case Realizations for the NextGen Iteration 20

- **Logging a Sale**
  - Once complete, the requirements state that the sale should be placed in an historical log.
  - Who is responsible for knowing all the logged sales and doing the logging? By Information Expert:
    - We can reasonably expect a Store to know all the logged sales since they are strongly related to its finances.
    - Other alternatives include classic accounting concepts, such as a SalesLedger. Using a SalesLedger object makes sense as the design grows and the Store becomes incohesive.
      - We would (ideally) add SalesLedger to the Domain Model as well since a sales ledger is a concept in the real-world domain. This kind of discovery and change during design work is to be expected.
Use Case Realizations for the NextGen Iteration 21

Store

addSale(s : Sale)

Store is responsible for knowing and adding completed Sales. Acceptable in early development cycles if the Store has few responsibilities.

SalesLedger

addSale(s : Sale)

SalesLedger is responsible for knowing and adding completed Sales. Suitable when the design grows and the Store becomes uncohesive.

Use Case Realizations for the NextGen Iteration 22

note that the Sale instance is named 's' so that it can be referenced as a parameter in messages 2 and 2.1
Use Case Realizations for the NextGen Iteration

• Calculating the Balance
  - The Process Sale use case implies that the balance due from a payment be printed on a receipt and displayed somehow
  - Because of the Model-View Separation principle, we should not concern ourselves with how the balance will be displayed or printed, but we must ensure that it is known
  - Who is responsible for knowing the balance?
    • To calculate the balance, we need the sale total and payment cash tendered. Therefore, Sale and Payment are partial Experts on solving this problem

Use Case Realizations for the NextGen Iteration

• If the Payment is primarily responsible for knowing the balance, it needs visibility to the Sale, to ask the Sale for its total. Since it does not currently know about the Sale, this approach would increase the overall coupling in the design—it would not support the Low Coupling pattern
• In contrast, if the Sale is primarily responsible for knowing the balance, it needs visibility to the Payment, to ask it for its cash tendered

{ bal = pmt.amount - s.total }

bal = getBalance

s : Sale

1: amt = getAmount

pmt: Payment

2: t = getTotal
Use Case Realizations for the NextGen Iteration

• How to Connect the UI Layer to the Domain Layer?
  - Common designs by which objects in the UI layer obtain visibility to objects in the domain layer include the following:
    • An initializer object (for example, a Factory object) called from the application starting method (e.g., the Java main method) creates both a UI and a domain object and passes the domain object to the UI.
    • A UI object retrieves the domain object from a well-known source, such as a factory object that is responsible for creating domain objects.
Use Case Realizations for the NextGen Iteration 27

Connecting the UI and domain layers

Use Case Realizations for the NextGen Iteration 28

- In the case of the enterItem message, we want the window to show the running total after each entry. Design solutions are:
  - Add a getTotal method to the Register. The UI sends the getTotal message to the Register, which delegates to the Sale
    • This has the possible advantage of maintaining lower coupling from the UI to the domain layer — the UI only knows of the Register object. But it starts to expand the interface of the Register object, making it less cohesive.
  - A UI asks for a reference to the current Sale object, and then when it requires the total (or any other information related to the sale), it directly sends messages to the Sale
    • This design increases the coupling from the UI to the domain layer. However, as we explored in the Low Coupling GRASP pattern discussion, higher coupling in and of itself is not a problem; rather, coupling to unstable things is a real problem
      - Assume we decide the Sale is a stable object
Use Case Realizations for the NextGen Iteration 29

- Initialization and the 'Start Up' Use Case
  - When to Create the Initialization Design?
    - Most systems have either an implicit or explicit Start Up use case and some initial system operation related to the starting up of the application
    - Although abstractly, a startUp system operation is the earliest one to execute, delay the development of an interaction diagram for it until after all other system operations have been considered.
      - This practice ensures that information has been discovered concerning the initialization activities required to support the later system operation interaction diagrams
    - Guideline: Do the initialization design last
Use Case Realizations for the NextGen Iteration

- How do Applications Start Up?
  - The startUp or initialize system operation of a Start Up use case abstractly represents the initialization phase of execution when an application is launched.
  - How an application starts and initializes depends on the programming language and operating system.
  - In all cases, a common design idiom is to create an initial domain object or a set of peer initial domain objects that are the first software "domain" objects created.
    - This creation may happen explicitly in the starting main method or in a Factory object called from the main method.
    - Often, the initial domain object (assuming the singular case), once created, is responsible for the creation of its direct child domain objects. For example, a Store chosen as the initial domain object may be responsible for the creation of a Register object.

- In Java, the main method may create the initial domain object or delegate the work to a Factory object that creates it.

```java
public class Main
{
    public static void main( String[] args )
    {
        // Store is the initial domain object.
        // The Store creates some other domain objects.
        Store store = new Store();
        Register register = store.getRegister();
        ProcessSaleJFrame frame = new ProcessSaleJFrame( register );
        ...
    }
}
```
Use Case Realizations for the NextGen Iteration

• Choosing the Initial Domain Object
  - What should the class of the initial domain object be?
  - **Guideline:** Choose as an initial domain object a class at or near the root of the containment or aggregation hierarchy of domain objects. This may be a facade controller, such as Register, or some other object considered to contain all or most other objects, such as a Store.
  - High Cohesion and Low Coupling considerations influence the choice between these alternatives.
    - In this application, we chose the Store as the initial object.

Use Case Realizations for the NextGen Iteration

• Store.create() Design
  - The tasks of creation and initialization derive from the needs of the prior design work, such as the design for handling enterItem and so on. By reflecting on the prior interaction designs, we identify the following initialization work: (next page)
    • Create a Store, Register, ProductCatalog, and ProductDescriptions.
    • Associate the ProductCatalog with ProductDescriptions.
    • Associate Store with ProductCatalog.
    • Associate Store with Register.
    • Associate Register with ProductCatalog.
  - **Multiplicity between classes of objects in the Domain Model and Design Model may not be the same**
    - The software Store object only creates one Register object, however, a real store may house many real registers or POS terminals.
Use Case Realizations for the NextGen Iteration

1.1: create
1.2.1*: put(id, price, description)
1.2.2*: put(id, pd)

the * in sequence number indicates the message occurs in a repeating section

The Command-Query Separation Principle

- Notice that the message to roll the Die is followed by a second getFaceValue to retrieve its new faceValue. In particular, the roll method is void — it has no return value.
The Command-Query Separation Principle 2

// style #1: used in the official solution
public void roll()
{
    faceValue = // random num generation
}

public int getFaceValue()
{
    return faceValue;
}

• Why not make roll non-void and combine these two functions so that the roll method returns the new faceValue, as follows?

// style #2; why is this poor?
public int roll()
{
    faceValue = // random num generation
    return faceValue;
}

The Command-Query Separation Principle 3

• You can find many examples of code that follow style #2, but it is considered undesirable because it violates the Command-Query Separation Principle, (CQS) a classic OO design principle for methods

• This principle states that every method should either be:
  - A command method that performs an action (updating, coordinating, ...), often has side effects such as changing the state of objects, and is void (no return value); or
  - A query that returns data to the caller and has no side effects — it should not permanently change the state of any objects

• The key point is that a method should not be both

• The roll method is a command — it has the side effect of changing the state of the Die's faceValue. Therefore, it should not also return the new faceValue, as then the method also becomes a kind of query and violates the "must be void" rule
The Command-Query Separation Principle

- **CQS** is widely considered desirable in computer science theory because with it,
  - You can more easily reason about a program’s state without simultaneously modifying that state.
  - It makes designs simpler to understand and anticipate.
    - For example, if an application consistently follows CQS, you know that a query or getter method isn’t going to modify anything and a command isn’t going to return anything. Simple pattern.
    - This often turns out to be nice to rely on, as the alternative can be a nasty surprise — violating the Principle of Least Surprise in software development.

The Command-Query Separation Principle

- Consider this contrived but explosive counter-example in which a query method violates CQS:

```java
Missile m = new Missile();
    // looks harmless to me!
...
String name = m.getName();
...
public class Missile {
    ...
    public String getName() {
        launch(); // launch missile!
        return name;
    }
} // end of class
```
Process: Iterative and Evolutionary Object Design

When
Near the beginning of each iteration, for a "short" period before programming.

Where
In a project room with lots of support for drawing and viewing drawings.

Who
Perhaps developers will do some design work in pairs. The software architect will collaborate, mentor, and visit with different design groups.

How: Tools
Software: A UML CASE tool that can also reverse engineer diagrams from code.

- Use two projectors attached to dual video cards.
- For whiteboard drawings, perhaps a digital camera.
- To print noteworthy diagrams for the entire team, a plotter for large-scale drawings to hang on walls.

Sample UP Artifacts and Timing

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Artifact</th>
<th>Incep.</th>
<th>Elab.</th>
<th>Const.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Modeling</td>
<td>Domain Model</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Iteration</td>
<td>I1</td>
<td>E1..En</td>
</tr>
<tr>
<td>Requirements</td>
<td>Use-Case Model (SSDs)</td>
<td>s</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Supplementary Specification</td>
<td>s</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Glossary</td>
<td>s</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td>Design Model</td>
<td>s</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SW Architecture Document</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Data Model</td>
<td>s</td>
<td>r</td>
<td></td>
</tr>
</tbody>
</table>

s - start; r - refine
Visibility Between Objects

- System operations illustrate messages between objects.
  - A sender object send a message to a receiver object, the sender must be visible to the receiver.
  - Sender must have some kind of reference or pointer to the receiver object.
  - e.g., the getProductDescription message sent from a Register to a ProductCatalog implies that the ProductCatalog instance is visible to the Register instance.
Visibility Between Objects

- Visibility from the Register to ProductCatalog is required

Visibility

- Visibility
  - Is the ability of an object to “see” or have a reference to another object.
  - One instance is within the scope of another.
- Visibility types (A sees B)
  - Attribute visibility — B is an attribute of A.
  - Parameter visibility — B is a parameter of a method of A.
  - Local visibility — B is a (non-parameter) local object in a method of A.
  - Global visibility — B is in some way globally visible.
Visibility 2

• Attribute visibility: B is an attribute of A.
  - It is a relatively permanent visibility because it persists as long
    as A and B exist.
  - A Register instance may have attribute visibility to a
    ProductCatalog, since it is an attribute of the Register.

```java
class Register {
    private ProductCatalog catalog;
    ...
    public void enterItem(itemID qty) {
        desc = catalog.getProductDes(itemID);
        ...}
}
```

Visibility 3

• Parameter Visibility: B is a parameter of a method of A
  - Parameter visibility from A to B exists when B is passed as a
    parameter to a method of A
  - It is a relatively temporary visibility because it persists only
    within the scope of the method.
  - When the makeLineItem message is sent to a Sale instance, a
    ProductDescription instance is passed as a parameter.

```java
makeLineItem(desc , qty)
```
Visibility 4

- It is common to transform parameter visibility into attribute visibility.
  - When the Sale creates a new SalesLineItem, it passes the ProductDescription in to its constructor.
  - The parameter is assigned to an attribute, thus establishing attribute visibility.

Visibility 5

- **Local Visibility** from A to B exists
  - when B is declared as a local object within a method of A.
  - It is a relatively temporary visibility.
- Two means by which local visibility is achieved
  - Create a new local instance and assign it to a local variable.
  - Assign the returning object from a method invocation to a local variable (e.g., enterItem method of class Register).
- As with parameter visibility, it is common to transform locally declared visibility into attribute visibility.
- A subtle version on the second variation is when the method does not explicitly declare a variable, but one implicitly exists as the result of a returning object from a method invocation
  ```java
  // there is implicit local visibility to the foo object
  // returned via the getFoo call
  anObject.getFoo().doBar();
  ```
Visibility 5

```java
enterItem(id, qty)
{
    ...
    // local visibility via assignment of returning object
    ProductDescription desc = catalog.getProductDesc(id);
    ...
}
```

Visibility 6

- **Global Visibility** from A to B exists
  - when B is global to A.
  - It is a relatively permanent visibility because it persists as long as A and B exist.
  - It is the least common form of visibility in object-oriented systems.
- To achieve global visibility
  - Assign an instance to a global variable, such as C++.
  - The preferred method to achieve global visibility is to use the Singleton pattern
Programming and Iterative, Evolutionary Development

• The prior design modeling should not be taken to imply that there is no prototyping or design-while-programming
  - Modern development tools provide an excellent environment to quickly explore and refactor alternate approaches, and some (often lots) design-while-programming is worthwhile

• A strength of use cases plus OOA/D plus OO programming is that they provide an end-to-end roadmap from requirements through to code
  - The road may not be smooth, but having a roadmap provides a starting point for experimentation and discussion
Programming and Iterative, Evolutionary Development

- Creativity and Change During Implementation
  - In general, the programming work is not a trivial code generation step
  - Realistically, the results generated during design modeling are an incomplete first step; during programming and testing, myriad changes will be made and detailed problems will be uncovered and resolved
  - The ideas and understanding (not the diagrams or documents!) Generated during OO design modeling will provide a great base that scales up with elegance and robustness to meet the new problems encountered during programming

- Mapping Designs to Code
  - Implementation in an object-oriented language requires writing source code for:
    - class and interface definitions
    - method definitions

Creating Class Definitions from DCDs

- DCDs depict the class or interface name, superclasses, operation signatures, and attributes of a class. This is sufficient to create a basic class definition in an OO language
- Defining a Class with Method Signatures and Attributes

```java
public class SalesLineItem {
    private int quantity;
    private ProductDescription description;
    public SalesLineItem(ProductDescription desc, int qty) { ... }
    public Money getSubtotal() { ... }
}
```

Constructor derived from the `create(desc, qty)` message sent to a `SalesLineItem` in the `enterItem` interaction diagram

The create method is often excluded from the class diagram because of its commonality and multiple interpretations, depending on the target language.
Creating Methods from Interaction Diagrams

- The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions.

- The enterItem message is sent to a Register instance; therefore, the enterItem method is defined in class Register.
  - public void enterItem(ItemID id, int qty)
- Message 1: A getProductDescription message is sent to the ProductCatalog to retrieve a ProductDescription.
  - ProductDescription desc = catalog.getProductDescription(id);
- Message 2: The makeLineItem message is sent to the Sale.
  - currentSale.makeLineItem(desc, qty);

The Register class

```java
public class Register {
    private ProductCatalog catalog;
    private Sale currentSale;
    public Register(ProductCatalog pc) {...}
    public void endSale() {...}
    public void enterItem(ItemID id, int qty) {...}
    public void makeNewItem() {...}
    public void makePayment(Money cashTendered) {...}
}
```
Creating Methods from Interaction Diagrams

```java
{ 
    ProductDescription desc = catalog.ProductDescription(id);
    currentSale.makeLineItem(desc, qty);
}
```

The `enterItem` method

Collection Classes in Code

- **One-to-many** relationships are common in DCD.
  - For example, a `Sale` maintains a group of `SalesLineItem` instances
  - In OO programming languages, these relationships are usually implemented with the introduction of a collection object, such as a `List` or `Map`, or even a simple array

```java
public class Sale {
    ...
    private List lineItems = new ArrayList();
}
```

- **Guideline:** If an object implements an interface, declare the variable in terms of the interface, not the concrete class.
Exceptions and Error Handling

• Exception handling has been ignored so far in the development of a solution.
  - This was intentional to focus on the basic questions of responsibility assignment and object design.
  - However, in application development, it’s wise to consider the large-scale exception handling strategies during design modeling (as they have a large-scale architectural impact), and certainly during implementation.
  - Briefly, in terms of the UML, exceptions can be indicated in the property strings of messages and operation declarations.

Defining the Sale.makeLineItem Method

• The makeLineItem method of class sale can also be written by inspecting the enterItem collaboration diagram.

```java
{ 
    lineItems.add(new SalesLineItem(desc, qty));
}
```
Order of Implementation

- Classes need to be implemented (and ideally, fully unit tested) from least-coupled to most-coupled
  - For example, possible first classes to implement are either Payment or ProductDescription; next are classes only dependent on the prior implementationsProductCatalog or SalesLineItem

Summary of Mapping Designs to Code

- There is a translation process
  - from UML class diagrams to class definitions
  - from interaction diagrams to method bodies.
- There is still lots of room for creativity, evolution, and exploration during programming work
- Sample code mapping for this NextGen POS iteration can be found in page 377-380
- Test-Driven Development (TDD)
  - An excellent practice promoted by the Extreme Programming (XP) method, and applicable to the UP and other iterative methods
  - In this practice, unit testing code is written before the code to be tested, and the developer writes unit testing code for all production code
  - The basic rhythm is to write a little test code, then write a little production code, make it pass the test, then write some more test code, and so forth
Chapter 21
Test-Driven Development and Refactoring

Introduction

- **Test-Driven Development (TDD)**
  - Extreme Programming (XP) promoted an important testing practice: writing the tests first.
  - An excellent practice promoted by the iterative and agile XP method, and applicable to the UP (as most XP practices are)
  - It also promoted continuously refactoring code to improve its quality - less duplication, increased clarity, and so forth
  - It is also known as test-first development.
  - TDD covers more than just unit testing (testing individual components),
  - But we will focus on its application to unit testing individual classes
Test-Driven Development

- **Key Point:** The test is written first, imagining the code to be tested is written.
- **Advantages include**
  - The unit tests actually get written
  - Programmer satisfaction leading to more consistent test writing
    - The test-last development, also known as Just-this-one-time-I'll-skip-writing-the-test development, doesn’t make developers feel satisfying
    - The psychological aspects of development can't be ignored — programming is a human endeavor.
  - Clarification of detailed interface and behavior
    - That reflection improves or clarifies the detailed design.
  - Provable, repeatable, automated verification
    - The early painful investment pays off as the size of the AP grows
  - The confidence to change things
    - When a developer needs to change existing code written by themselves, there is a unit test suite that Can be run, providing immediate feedback if the change caused an error.

Test-Driven Development

- The popular unit testing framework is the xUnit family
  - Java: JUnit; .NET: NUnit
  - JUnit is integrated into most of the Java IDEs, such as Eclipse.
- Example of Sale class, write a unit testing method in a SaleTest class
  - Create a Sale—the thing to be tested (also known as the fixture).
  - Add some line items to it with the makeLineItem method (the makeLineItem method is the public method we wish to test).
  - Ask for the total, and verify that it is the expected value, using the assertTrue method. JUnit will indicate a failure if any assertTrue statement does not evaluate to true.
- Each testing method follows this pattern:
  - Create the fixture.
  - Do something to it (some operation that you want to test).
  - Evaluate that the results are as expected.
- A key point to note is that we do not write all the unit tests for Sale first; rather, we write only one test method, implement the solution in class Sale to make it pass, and then repeat.
Test-Driven Development

• **TDD Rhythm**
  - Quickly add a test
  - Run all tests and see the new one fail
  - Make a little change
  - Run all tests and see them all pass
  - Refactor to remove duplication

  **Do it as quickly as possible**

  **Red**

  **Green**

  **Refactor**

  **Takes time, but keeps green**

Test-Driven Development

• To use JUnit, to create a test class that extends the JUnit TestCase class; the test class inherits various unit testing behaviors.
  - create a separate testing method for each Sale method.
  - write unit testing methods for each public method of the Sale class.

• To test method doFoo, it is an idiom to name the testing method testDoFoo.

```java
public class SaleTest extends TestCase {
    // …
    // test the Sale.makeLineItem method
    public void testMakeLineItem() {
        // STEP 1: CREATE THE FIXTURE
        // -this is the object to test
        // -it is an Idiom to name it 'fixture'
        // -it is often defined as an instance field rather than
        // a local variable Sale fixture = new Sale();
        // set up supporting objects for the test
        Money total = new Money( 7.5 );
        Money price = new Money( 2.5 );
    }
}
```
Test-Driven Development 4

ItemID id = new ItemID(1);
ProductDescription desc = new ProductDescription(id, price, "product 1");

// STEP 2: EXECUTE THE METHOD TO TEST
// NOTE: We write this code "imagining" there
// is a makeLineItem method. This act of imagination
// as we write the test tends to improve or clarify
// our understanding of the detailed interface to
// to the object. Thus TDD has the side-benefit of
// clarifying the detailed object design.
// test makeLineItem
sale.makeLineItem(desc, 1);
sale.makeLineItem(desc, 2);

// STEP 3: EVALUATE THE RESULTS
// there could be many assertTrue statements
// for a complex evaluation
// verify the total is 7.5 (2.5 + 2.5*2)
assertTrue(sale.getTotal().equals(total));

Test-Driven Development 5

- Important: always automate testing
  - If you need to know whether a particular class is correctly written, write a unit test
  - If you need to see something, write a unit test
  - If you need to debug a particular class, write a unit test.

```java
class Foo {
  ...
}

void main() {
  Foo f(...);
f.SetXXX(...);
f.DoXXX(...);
cout << f.GetXXX(...);
}

class Foo {
  ...
}

void TestFooXXX() {
  Foo f(...);
f.SetXXX(...);
f.DoXXX(...);
AppDelegate(f.GetXXX(), ...);
}
```
IDE Support for TDD and xUnit

Refactoring

- **Refactoring**
  - Is a structured, disciplined method to rewrite or restructure existing code without changing its external behavior each step.

- **The essence of refactoring**
  - Applying small behavior preserving transformations (each called a 'refactoring'), one at a time.
  - After each transformation, the unit tests are re-executed to prove that the refactoring did not cause a regression (failure).
  - Each refactoring is small, but a series of transformations — each followed by executing the unit tests again — can, of course, produce a major restructuring of the code and design, all the while ensuring the behavior remains the same.
Refactoring 2

- The activities and goals of refactoring
  - Remove duplicate code
  - Improve clarity
  - Make long methods shorter
  - Remove the use of hard-coded literal constants
- Code that’s been well-refactored is short, tight, clear, and without duplication — it looks like the work of a master programmer.
- Code that doesn’t have these qualities smells bad or has code smells
  - Code smells is a metaphor in refactoring — they are hints that something may be wrong in the code
  - Code stench — truly putrid code that requires for clean up

Refactoring 3

- Some code smells include
  - Duplicated code
  - Big method
  - Class with many instance variables
  - Class with lots of code
  - Strikingly similar subclasses
  - Little or no use of interfaces in the design
  - High coupling between many objects
- The remedy to smelly code are the refactorings.
- Like patterns, refactorings have names, such as Extract Method.
  - There are about 100 named refactorings
Refactoring 4

- Examples of Refactorings

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Method</td>
<td>Transform a long method into a shorter one by factoring out a portion into a private helper method.</td>
</tr>
<tr>
<td>Extract Constant</td>
<td>Replace a literal constant with a constant variable.</td>
</tr>
<tr>
<td>Introduce Explaining Variable</td>
<td>Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.</td>
</tr>
<tr>
<td>Replace Constructor Call with Factory Method</td>
<td>In Java, for example, replace using the new operator and constructor call with invoking a helper method that creates the object (hiding the details).</td>
</tr>
</tbody>
</table>

Refactoring 5

- Example of the Extract Method refactoring
  We can make the takeTurn method shorter, clearer, and better supporting High Cohesion by extracting that code into a private helper method called rollDice

```java
public class Player
{
    private Piece piece;
    private Board board;
    private Die[] dice;
    // ...
    public void takeTurn()
    {
        // roll dice
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }
        // move to the target square
        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
    }
} // end of class
```
Refactoring 6

- The code after applying the Extract Method refactoring

```java
public class Player
{
    private Piece piece;
    private Board board;
    private Die[] dice;
    // ...

    public void takeTurn()
    {
        // the refactored helper method
        int rollTotal = rollDice();
        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
    }

    private int rollDice()
    {
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
            dice[i].roll();
            rollTotal += dice[i].getFaceValue();
        }
        return rollTotal;
    }
}
} // end of class
```

Refactoring 7

- The Introduce Explaining Variable refactoring
  - It clarifies, simplifies, and reduces the need for comments
  - Before introducing an explaining variable.
    // good method name, but the logic of the body is not clear

```java
boolean isLeapYear( int year ) {
    return( ( ( year % 400 ) == 0 ) || ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) );
}
```

- After introducing an explaining variable.
  // that's better!

```java
boolean isLeapYear( int year ) {
    boolean isFourthYear = ( ( year % 4 ) == 0 );
    boolean isHundrethYear = ( ( year % 100 ) == 0 );
    boolean is4HundrethYear = ( ( year % 400 ) == 0 );
    return(is4HundrethYear || (isFourthYear && !isHundrethYear));
}
```
IDE Support for Refactoring: Eclipse IDE 1

```java
public Player(String name, Die[] dice, Board board) {
    this.name = name;
    this.dice = dice;
    this.board = board;
    piece = new Piece(board.getStartSquare());
}

public void takeTurn() {
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++) {
        dice[i].roll();
        rollTotal += dice[i].getFaceValue();
    }
    Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);
}
```

IDE Support for Refactoring: Eclipse IDE 2

```java
public void takeTurn() {
    int rollTotal = rollDice();

    Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
    piece.setLocation(newLoc);
}

private int rollDice() {
    // roll dice
    int rollTotal = 0;
    for (int i = 0; i < dice.length; i++) {
        dice[i].roll();
        rollTotal += dice[i].getFaceValue();
    }
    return rollTotal;
}
```
The Book

Refactoring
Improving the Design of Existing Code

Martin Fowler
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.