T3a

Introduction to Design Patterns

Code Libraries vs Frameworks

- Code (or component) libraries:
  - consist of a set of related and reusable classes
  - do not impose a design on an application
- Frameworks:
  - constitute a reusable design for a whole application or subsystem
  - consist of a set of cooperating classes, which
    - » identifies partitioning of responsibility
    - » implements collaborations
  - are customized through sub-classing
  - are targeted towards a narrow application area
- We want to reuse without committing to a particular implementation!!

Enter Design Patterns

- Design patterns aim to directly facilitate the design process:
  - provide ready-made solutions to small design problems that are part of the larger application or subsystem design,
  - refine rather than reinvent,
  - record experience in designing object-oriented software in a form that people can use effectively,
  - are a solution to a design problem in context — "a description of communicating objects and classes that are customized to solve a general design problem." [Gamma94]

Dealing with Change

- Fundamental principle: Anticipate requirement evolution and design for it by localizing its effects in your design.
  - Understand variability and likelihood of change
  - Handle change by committing to decoupling, typically via the principle of locality of change
- In choosing a Design Pattern you must first understand:
  - the purpose of each pattern:
    - » what forces are resolved
  - how patterns interrelate:
    - » Several patterns may solve the same design problem
      - The selection problem

Overview of Go4 Design Patterns

(book of Gamma, Helm, Johnson and Vlissides)

The Patterns of Gamma et al.

Motivation:

- Design Patterns record experience of designers solutions to OO problems which occur often
- Patterns capture design expertise
- Gamma's patterns deal with general OO design problems and proven solutions
  - Each pattern focuses on a particular OO design problem or issue
- Gamma's patterns are not domain specific and do not address concurrency, distribution, real-time programming, GUIs, and databases.

Scope of Gamma's Design Patterns

"One person's pattern is another's idiom"

- No patterns for linked lists, stacks, hash tables, …
- No domain specific patterns
- No architectural patterns
- Assumes features common in OO languages:
  - No language specific patterns (e.g., private derivation) in theory
  - Inheritance and polymorphism used for subtyping
  - A pattern for Smalltalk may be trivial in C++, or vice versa
  - "A language constrains what we can think about"

Pattern Format

- **Pattern Name and Classification**
- **Intent** -pattern's intent, issues addressed
- **Also Known As** -other known names for pattern
- **Motivation** -scenario illustrating design problem
- **Applicability**-situations where pattern applies
- **Structure** -OMT/UML style picture of pattern's structure
- **Participants** -object, classes, and their responsibilities
- **Collaborations** -how participants collaborate
- **Consequences** -tradeoffs, what can be varied
- **Implementation** -hints, techniques, pitfalls
- **Sample Code** -C++ or Smalltalk sample code
- **Known Uses** -from real systems
- **Related Patterns** -other closely related patterns

Sections of the Pattern Format

- Pattern Name **forms a basic design vocabulary**
- Problem **describes when to apply a pattern**
- Solution **describes the solution elements, their responsibilities, relationships, and collaborations.**
  - The solution is abstract, not implementation dependent
- Motivation **and** Applicability **sections illustrate and define respectively the context of a design pattern and contain links back to requirements, design constraints and analysis.**
- Structure, Participants **and** Collaborations **sections describe the design with emphasis on collaborations and contracts fulfilled by the participants.**
- Consequences **section deals with evaluating the approach taken by the pattern.**
  - **linked back to flexibility requirements, and also to performance evaluation aspects.**
- Implementation **section discusses details of translating the pattern into code.**
- Sample code **is an illustration of a mapping of the design pattern into code.**

Guideline to Using a Pattern

- **Read the pattern once through for an overview.**
- **Go back and study the Structure, Participants, and Collaborations sections.**
- **Look at the Sample Code section to see a concrete example of the pattern in code.**
- **Choose names for pattern participants that are meaningful in your application context.**
- **Define the relevant classes in your context.**
- **Define application-specific names for operations in the pattern.**
- **Implement the operations to carry out the responsibilities and collaborations in the pattern.**

Gamma's Patterns (Names and Classification)

Purpose

	Creational	Structural	Behavioural
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

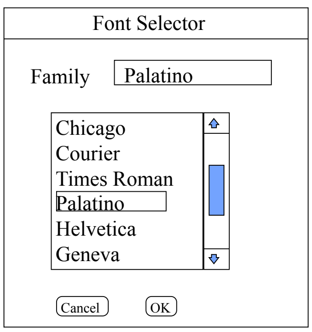
Three Key Behavioral Patterns

Behavioral Patterns Overview

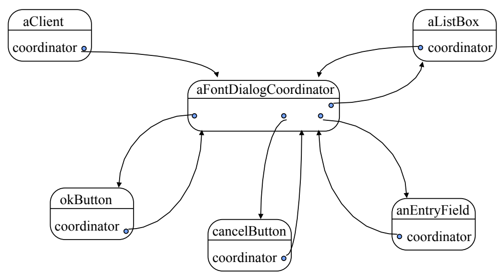
- **Designs that deal with partitioning of behavioral responsibilities**
  - control flow tends to be hard to follow.
- **Behavioral class patterns: use inheritance to partition responsibilities**
  - E.g., Template Method
- **Behavioral object patterns: use object composition to partition behavior**
  - E.g., Mediator, Observer, State, Strategy, Iterator

The Mediator Pattern

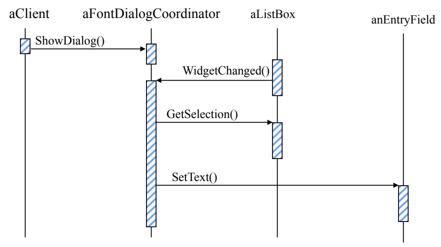
Example



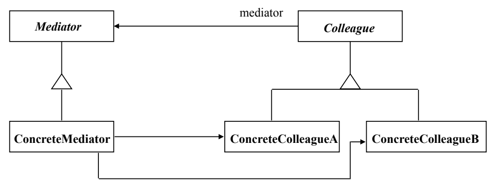
Motivation



Collaborations



Structure



Consequences

- limits subclassing
- decouples colleagues
- simplifies object protocols
- abstracts how objects collaborate
- centralizes control

Mediator Sample Code (1)

```
class DialogDirector {
public:
    virtual ~DialogDirector();              //destructor
    virtual void ShowDialog();
    virtual void WidgetChanged(Widget*) = 0;
protected:
    DialogDirector();                       //constructor
    virtual void CreateWidgets() = 0;
};
```

Mediator Sample Code (2)

```
class FontDialogDirector : public DialogDirector {
public:
   FontDialogDirector();                  // constructor
   virtual ~FontDialogDirector();         // destructor
   virtual void WidgetChanged(Widget*);   // arg. is a pointer to Widget
protected:
   virtual void CreateWidgets();
private:                                  // attributes of this
   Button* _ok;                           // specific Dialog
   Button* _cancel;
   ListBox* _fontList;
   EntryField* _fontName;
};
```


Mediator Sample Code (3)

```
void FontDialogDirector::CreateWidgets () {
// code to create this specific Dialog
// this passes the current receiver to the components it builds
   _ok = new Button(this);
   _cancel = new Button(this);
   _fontList = new ListBox(this);
   _fontName = new EntryField(this);

   // fill the listBox with the available font names

   // assemble the widgets in the dialog
}
```


Mediator Sample Code (4)

```
void FontDialogDirector::WidgetChanged (Widget* theChangedWidget) {
// this method handles a change
// its argument is the widget that has changed
   if (theChangedWidget == _fontList) {
      _fontName->SetText(_fontList->GetSelection());

   } else if (theChangedWidget == _ok) {
      // apply font change and dismiss dialog
      // ...
   } else if (theChangedWidget == _cancel) {
      // dismiss dialog
   }
}
```


Mediator Sample Code (5)

```
class Widget {
public:
// The constructor requires being told about the DialogDirector.
// By subclass substitution, this pointer can be to any subclass
// of DialogDirector.
   Widget(DialogDirector*);
   virtual void Changed();
   virtual void HandleMouse(MouseEvent& event);
   // ...
private:
   DialogDirector* _director;
};
void Widget::Changed () {
   _director->WidgetChanged(this);
}
```

Mediator Sample Code (6)

```
class Button : public Widget {
public:
   Button(DialogDirector*);

   virtual void SetText(const char* text);
   virtual void HandleMouse(MouseEvent& event);
   // ...
};
void Button::HandleMouse (MouseEvent& event) {
   // ...
   Changed();   // see code in class Widget
}
```


Mediator Sample Code (7)

```
class ListBox : public Widget {
public:
   ListBox(DialogDirector*);
   virtual const char* GetSelection();
   virtual void SetList(List<char*>* listItems);
   virtual void HandleMouse(MouseEvent& event);
   // ...                                              };
class EntryField : public Widget {
public:
   EntryField(DialogDirector*);
   virtual void SetText(const char* text);
   virtual const char* GetText();
   virtual void HandleMouse(MouseEvent& event);
   // ...                                              };
```


The Observer Pattern

Intent

Observer Pattern:

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its *dependents* are notified and updated automatically

(Notice the obvious usefulness to GUI applications)

Also known as: Dependents, Publish-Subscribe

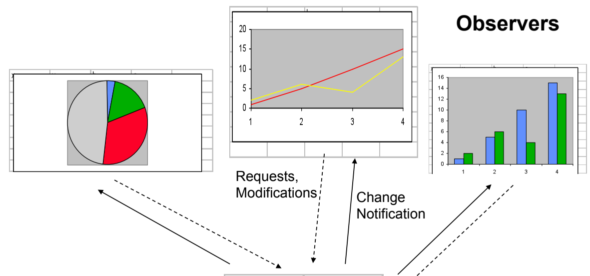
Motivation

Motivation:

"Common side effect of breaking a system into a collection of cooperating classes is the need to maintain consistency between related objects.

You don't want to achieve consistency by making classes tightly coupled, this would increase complexity and reduce reusability."

Observers

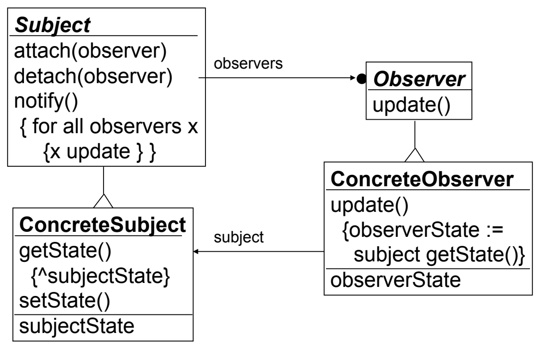


x	y
1	2
5	6
10	4
15	13

Subjects and Observers

- **A subject will notify its observers (dependents) whenever this subject changes state.**
- **Subjects don't know or care who their observers are.**
- **Observers will *register* interest in the subject, and query the subject for state information when notified of a state change.**
  - **Registration and deregistration are dynamic operations**

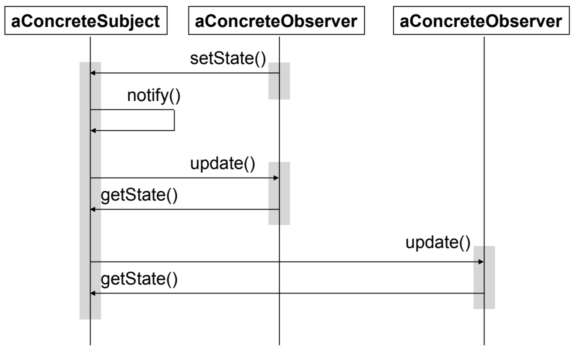
Structure



Participants

- **Subject**
  - knows it has some number of observers
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject
- **Concrete Subject**
  - stores state of interest
  - notifies observers whenever a change occurs that could leave some observer inconsistent
- **Concrete Observer**
  - maintains reference to concrete subject
  - stores a state that should be consistent with the one of its subject
  - implements the Observer updating interface to keep its state consistent with the one of its subject

Collaborations



Consequences

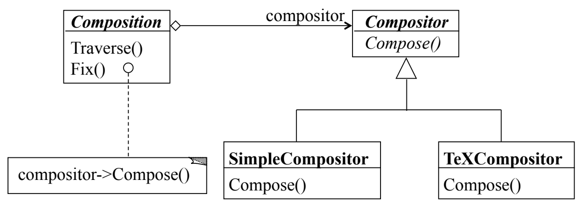
- **Subjects and observers can be varied or reused independently of each other**
- **Abstract coupling between subject & observer**
  - subject only knows it has some observers, not who they are
- **Support for broadcast-style communication:**
  - notification of change does not have a specific receiver
    - » it is sent to all interested parties
  - observers can be added/deleted at any time
- **Unexpected Updates**
  - observers don't know about each other
    - » a seemingly innocent action on a subject can cause a cascade of (often spurious) updates
    - » the problem of redundant updates is aggravated by the fact that the update protocol does *not* indicate what changed

Some Implementation Issues

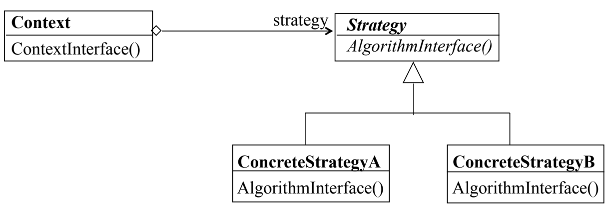
- **Mapping subjects to observers:**
  - simple strategy: keep references in subject, but this is wasteful if there are many subjects and few observers
  - alternative: keep separate subject-to-observers tables
- **Observing more than one subject:**
  - we must modify the update protocol to indicate which subject is notifying...
- **Notifying:**
  - should the subject notify or the observers poll?
  - should the notification hold information of the subject?
- **Deleting:**
  - we must avoid dangling references!
    - » the deletion protocol must have the subject notify observers before disappearing
    - » similarly, an observer cannot just die

The Strategy Pattern

Strategy: Example



Strategy: Structure



Strategy: Collaborations

- **Interaction between Context and Strategy**
  - **Context passes data to the strategy**
  - **Context passes itself to the strategy**
- **Context services client requests by using the Strategy:**
  - **The Strategy may be chosen by the client, but the client only interacts with the context.**
  - **There may be a family of ConcreteStrategies for the client to chose from.**

Strategy: Consequences

- Applies to families of related algorithms.
- Avoids subclassing contexts.
- Avoids conditionals for selecting one of several behaviors.
- Permits selection of implementations of same kind of behavior.
- Clients need to know about the strategies.
- Introduces a communication overhead between Strategy and Context.
- Requires more objects than a simpler solution.

Strategy: Implementation

- On the interface between Strategy and Context
  - a ConcreteStrategy needs efficient access to a Context's data,
  - a ConcreteStrategy needs to pass data to the context ,
  - a Context may have to be passed to a strategy:
    - » Passing data to the strategy decouples context from the strategy but may lead to passing data not required by the strategy
    - » Having a context pass itself to the strategy requires that a context have a separate interface specifically for letting the strategy access context's data.