SWE30003
Software Architectures and Design

Lecture 7
Patterns in Software Development

SWINBURNE
UNIVERSITY OF
TECHNOLOGY

1

Logistical matters

- Weekly submissions – A & Q
    - Week 2: 362 and 341 out of 459;
    - Week 3: 397 and 375 out of 459;
    - Week 4: 399 and 390 out of 459;
    - Week 5: 380 and 372 out of 453;
    - Week 6: 389 and 382 out of 453;
    - Week 7:
    - Week 8:
    - This is a hurdle requirement, no late submission

2

2

Question to Answer – week 6

Consider the situation of designing a Chess application. Would you model a Queen to be a multiple heir of Rook and Bishop? If so, why? If not, why not and how would you model the commonalities of Queen, Rook and Bishop? Which heuristic(s) can help you in making a decision?
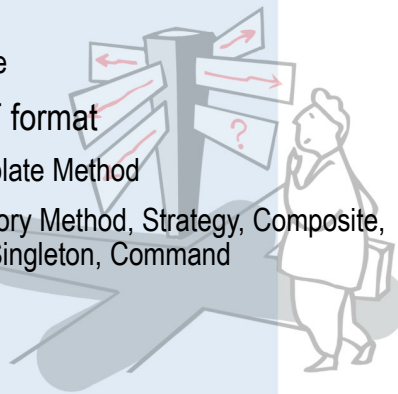
3

3

Outline

- What are Patterns?
- Idioms
    - ☐ Delegation and Interface
- Design Patterns — GOF format
    - ☐ Detailed example: Template Method
    - ☐ Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command
- Other kinds of Patterns
    - ☐ Pattern systems

4

4

Principal References

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns,* Addison-Wesley, 1995.

- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns,* Addison-Wesley, 1996.

5

5

Problem solving is pattern matching!
(to a large extend)

6

6

Outline

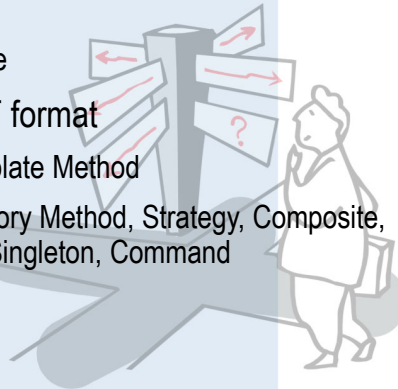- **What are Patterns?**
- Idioms
  - □ Delegation and Interface
- Design Patterns — GOF format
  - □ Detailed example: Template Method
  - □ Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command
- Other kinds of Patterns
  - □ Pattern systems

7

What are Patterns?

Patterns were first systematically catalogued in the domain of (building) architecture:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

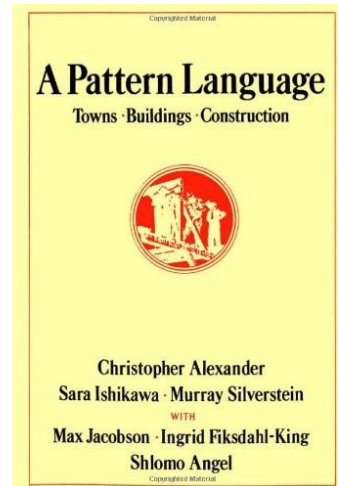— Alexander et al., *A Pattern Language*, 1977

8

A Pattern Language

- The "original" book about the idea of using a generic solution/scheme to a reoccurring problem in the design of artifacts (buildings).

A Pattern Language
Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

9

9

(Software) Design Patterns

Design patterns document standard <u>solutions</u> to common (software) design <u>problems</u>:

"Each pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively."
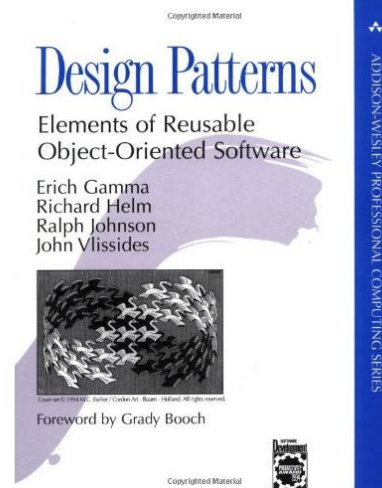
— Gamma et al., Design Patterns, 1995

10

10

Design Patterns (cont.)

- Seminal book that describes the notion of patterns in the context of object-oriented design

- A **MUST** book for a software designer!

Design Patterns
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

11

11

What Design Patterns are not…

Algorithms are not software design patterns:

- algorithms solve *computation* problems, not *design* problems,

- merge-sort is an algorithm; divide and conquer is a (design) pattern.

Software components are not software design patterns:

- design patterns describe a ***way*** of solving a problem,

- design patterns document pros and cons of different implementations,

- software components are *specific **implementations***, and may be *implemented* using design patterns.

- Type (vs-instances) vs. "entities"

12

12

What Design Patterns are not (cont.)

Frameworks are not software design patterns:

- a framework ***implements*** a "*generic*" *software architecture*,

- a design pattern <u>documents</u> the solution to a specific <u>design problem</u>,

- a framework may *use and be documented with* design patterns,

- like frameworks, design patterns are drawn from experience with multiple applications solving related problems.

13

13

What Problems do Patterns solve?

Software design patterns <u>document</u> *design experience:*

- Patterns enable *widespread reuse* of software structures,

- Patterns improve *communication* within and across software development teams,

- Patterns explicitly *capture knowledge* that experienced developers already understand implicitly,

- Useful patterns arise from *practical experience*,

- Patterns *facilitate training* of new developers,

- Patterns help to transcend "programming language-centric" viewpoints.

— Doug Schmidt, CACM, Oct. 1995

14

14

What are Idioms and Patterns?

Idioms	Idioms are *common programming techniques* and conventions. They are often *language-specific*.
Patterns	Patterns document *common solutions to design problems*. They are *language-independent*.
Libraries	Libraries are *collections of functions, procedures or other software abstractions* that can be used in many applications.
Frameworks	Frameworks are <u>open libraries</u> that define the *generic architecture* of an application, and can be <u>extended</u> by adding or deriving new classes.

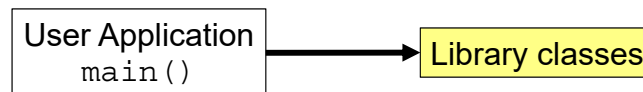***Frameworks typically make use of common idioms and patterns.***

15

15

Frameworks vs. Libraries

- In traditional application architectures, user code makes use of library functionality in the form of procedures or classes:

  User Application `main()` → Library classes

- A framework *reverses* the usual relationship between generic and application code. Frameworks provide both generic functionality and application architecture:

  User classes ← Framework classes `main()`

- Essentially, a framework says: *"Don't call me — I'll call you."*
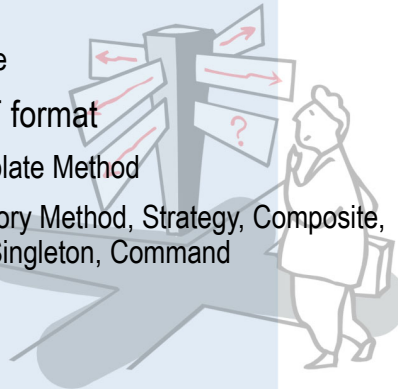
16

16

Outline

- ■ What are Patterns?
- ■ **Idioms**
  - □ Delegation and Interface
- ■ Design Patterns — GOF format
  - □ Detailed example: Template Method
  - □ Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command
- ■ Other kinds of Patterns
  - □ Pattern systems

17

17

Idioms

"An idiom is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language."

— Buschmann et al, Pattern-Oriented Software Architecture, 1995

For a collection of sample idioms: have a look at:

James O. Coplien, *Advanced C++: Programming Styles and Idioms*, Addison-Wesley, 1992

18

18

Idiom: Delegation

Problem: <u>How can an object share behavior without inheritance?</u>

Solution: *Delegate some of its work to another object*

☐ Inheritance is a common way to extend the behavior of a class, but can be an inappropriate way to combine features.

☐ Delegation *reinforces encapsulation* by keeping roles and responsibilities distinct.

19

19

Delegation

Example

■ When a `ChessBoard` is asked to `undo` a move, it delegates the work to the corresponding `Move` object.

Consequences

■ More *flexible, less structured* than inheritance.

☞ Delegation is one of the *most basic object-oriented idioms*, and is used by almost all design patterns.

20

20

Delegation example

```
public class ChessBoard {
  ...
  public void undoMove() {
    if (!(this.moves.isEmpty())) {
      Move lastMove = this.moves.lastElement();
      lastMove.undo (this);
      this.moves.removeElement(lastMove);
    }
  }
}
```

NOTE: `this` as parameter to undo!

21

21

Idiom: Interface

Problem: How do you keep a client of a service independent of classes that provide the service?

Solution: *Have the client use the service through an interface rather than a concrete class.*

- ☐ If a client names a concrete class as a service provider, then only instances of that class or its subclasses can be used in future.

- ☐ By naming an interface, an instance of *any class* that implements the interface can be used to provide the service.

22

22

Interface

Example

■ Any class that implements the `Player` interface may be used as valid chess player.

Consequences

■ Interfaces *reduce coupling* between classes.

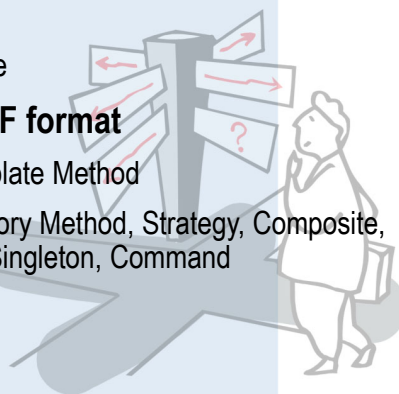■ They also *increase complexity* by adding *indirection*.

23

23

Outline

■ What are Patterns?

■ Idioms

  □ Delegation and Interface

■ **Design Patterns — GOF format**

  □ Detailed example: Template Method

  □ Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command

■ Other kinds of Patterns

  □ Pattern systems

24

24

Design Patterns

"A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context."

— Gamma et al., Design Patterns, 1995

25

25

Pattern Format

- **Pattern Name and Classification:** should convey the *essence* of a pattern
  - □ *Also Known As:* other common names
- **The Problem Forces:** describes when to apply the pattern
  - □ *Intent:* short statement of rationale and intended use
  - □ *Motivation:* a problem scenario and example solution
  - □ *Applicability:* in which situations can the pattern be applied

— Gamma et al., Design Patterns, 1995

26

26

Pattern Format (cont.)

- **The Solution:** *abstract* description of design elements
  - ☐ Structure: class and scenario diagrams
  - ☐ Participants: participating classes/objects and their responsibilities
  - ☐ Collaborations: how participants carry out responsibilities
- **The Consequences:** results and trade-offs of applying the pattern
  - ☐ Implementation: pitfalls, hints, techniques, language issues etc.
  - ☐ Sample Code: illustrative examples in various programming languages
  - ☐ Known Uses: examples of the pattern found in real systems
  - ☐ Related Patterns: competing and supporting patterns

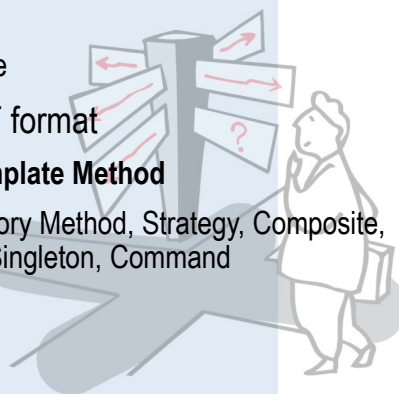— Gamma et al., Design Patterns, 1995

27

27

Outline

- ■ What are Patterns?
- ■ Idioms
  - ☐ Delegation and Interface
- ■ Design Patterns — GOF format
  - ☐ **Detailed example: Template Method**
  - ☐ Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command
- ■ Other kinds of Patterns
  - ☐ Pattern systems

28

28

Pattern: Template Method

(Adapted from Gamma et al., Design Patterns, pp. 325-330)

Name:

Template Method
— *Classification: Class Behavioural*

Intent:

"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

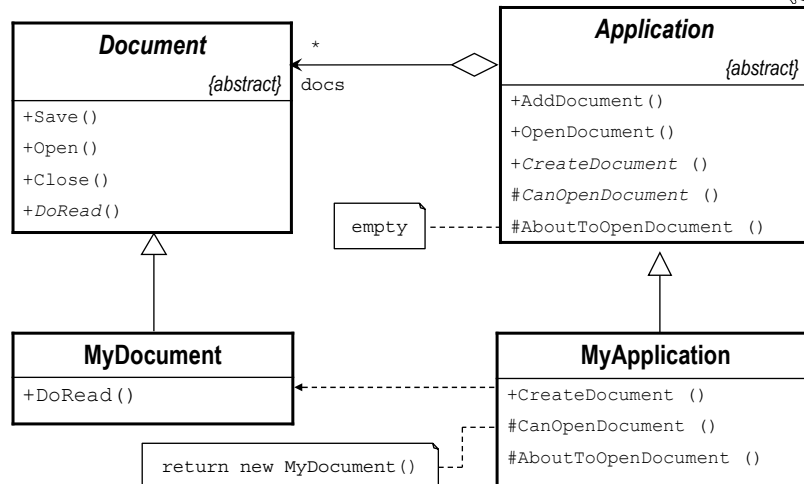29

29

Template Method — Motivation

- An application framework provides `Application` and `Document` classes. `Application` is responsible for opening existing documents stored in an external format. An opened document is represented by a `Document` instance.

- An application built with the framework should subclass `Application` and `Document` for specific kinds of documents.

30

30

Template Method – Motivation (cont.)

Document *{abstract}*
+Save()
+Open()
+Close()
+*DoRead*()

* docs

Application *{abstract}*
+AddDocument()
+OpenDocument()
+*CreateDocument* ()
#*CanOpenDocument* ()
#AboutToOpenDocument ()

empty

MyDocument
+DoRead()

MyApplication
+CreateDocument ()
#CanOpenDocument ()
#AboutToOpenDocument ()

return new MyDocument()

31

31

Template Method – Motivation (cont.)

The abstract Application class defines the algorithm for opening and reading a document in its `OpenDocument` operation:

```
void Application::OpenDocument (const char* name)
{
  if (!CanOpenDocument(name)) {    // can the document be opened?
    return;
  }
  Document* doc = CreateDocument(name);
  if (doc) {                       // successful creation
    _docs->AddDocument(doc);
      AboutToOpenDocument(doc);  // warn Application subclass instances
    doc->Open();
    doc->DoRead();
  }
}
```

NOTE: a `Document` instance *reads itself*…

32

32

Template Method – Motivation (cont.)

- `OpenDocument` is a *template method*, since it defines an algorithm in terms of abstract operations that subclasses *override* to provide concrete behaviour.

- *Subclasses* must provide the logic for `CanOpenDocument` and `CreateDocument`.

- If special actions are needed to prepare for opening documents, they may be specified by overriding `AboutToOpenDocument`.

33

33

Template Method – Applicability
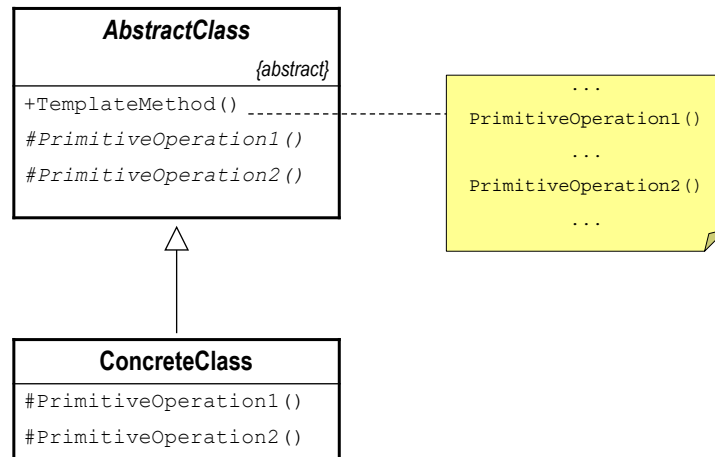
Applicability:

The Template Method should be used:

- to implement the *non-varying parts of an algorithm* once and allow subclasses to implement the parts that may vary.

- to *refactor* common behaviour among subclasses into a common superclass.
    - ☞ Comment: this is a good example of "refactoring to generalize."

- to *control subclass extensions*. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

34

34

Template Method – Structure

AbstractClass

{abstract}

+TemplateMethod()

#PrimitiveOperation1()

#PrimitiveOperation2()

```
        ...
PrimitiveOperation1()
        ...
PrimitiveOperation2()
        ...
```

ConcreteClass

#PrimitiveOperation1()

#PrimitiveOperation2()

35

35

Template Method – Participants

Participants

- **AbstractClass** (e.g., `Application`)
  - □ declares abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - □ defines a template method that implements the skeleton of an algorithm.
    The template method calls the primitive operations as well as operations defined in `AbstractClass` or elsewhere.

- **ConcreteClass** (e.g., `MyApplication`)
  - □ implements the primitives operations to carry out subclass-specific steps of the algorithm

36

36

Template Method – Collaborations

Collaborations

- ■ `ConcreteClass` relies on `AbstractClass` to implement the non-varying steps of the algorithm

37

37

Template Method – Consequences

Consequences

- ■ Template methods are a fundamental technique for *factoring out common behaviour* in class libraries.

- ■ They lead to an *inverted control structure* since a parent class calls the operations of a subclass and not the other way around.

38

38

Template Method – Consequences (II)

Template methods tend to call one of several kinds of operations:

- concrete operations (on client classes),
- concrete `AbstractClass` operations,
- primitive operations (i.e., declared abstract in `AbstractClass`),
- factory methods (i.e., abstract operations for creating objects),
- hook operations that subclasses can extend.

☞ It is important for template methods to specify which operations are hooks (*may* be overridden) and which are *abstract* operations (*must* be overridden).

39

39

Template Method – Known Uses

Known Uses

- Template methods are so fundamental that they can be found in almost every abstract class.

Related Patterns

- *Factory Methods* are often called by template methods. In the Motivation example, the factory method `CreateDocument` is called by the template method `OpenDocument`.
- *Strategy*: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.
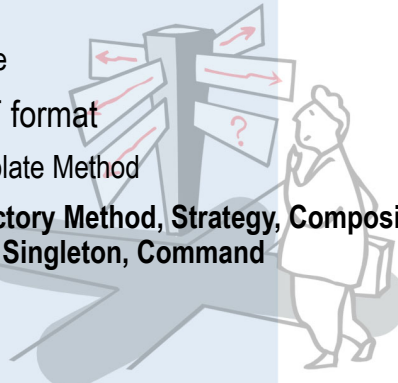
40

40

Outline

- What are Patterns?
- Idioms
  - ☐ Delegation and Interface
- Design Patterns — GOF format
  - ☐ Detailed example: Template Method
  - ☐ **Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command**
- Other kinds of Patterns
  - ☐ Pattern systems

41

41

Pattern: Factory Method

Problem: How to define an interface to create objects, but defer the decision which specific class to instantiate?

Solution: *Introduce a Factory Method in a base class (or interface) and delegate the decision which concrete class(es) to instantiate to subclasses (or implementing classes).*

- Factory Methods are often used in frameworks to properly encapsulate the instantiation of a derivations of a base abstraction.

42

42

Factory Method - Chess Example

```java
public abstract class ChessPiece {
    ...
    protected Collection<Behaviour> behaviours = null;
    // Constructor
    protected ChessPiece(...) {
        ...
        this.behaviours = new HashSet<Behaviour>();
        // call a deferred method
        this.setBehaviours();
    }
}
    ...
    // Instantiations of concrete behaviors is left to subclasses
    protected abstract void setBehaviours();
}
```

43

43

Pattern: Strategy

Problem: How to define a family of algorithms and make them interchangeable, independent of clients that may use them?
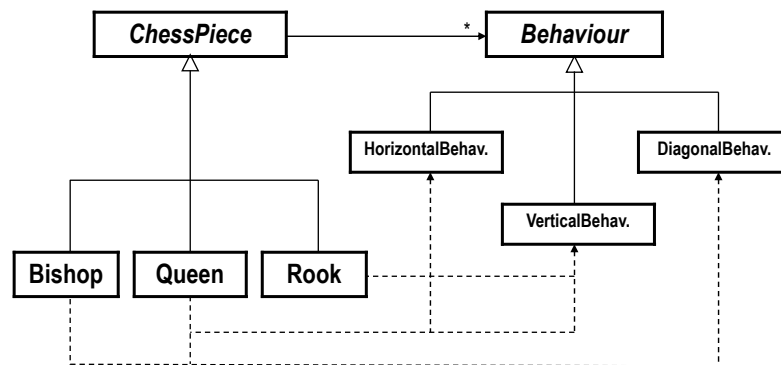
Solution: *Define a common interface for the family of algorithms and have specific implementations implement this interface.*

■ Strategies are used in conjunction with template methods to factor out common behavior of varying algorithms.

44

44

Strategy - Chess Example

45

45

Strategy - Chess Example (cont.)

```java
public abstract class ChessPiece {
    ...
    public boolean isValid (Move aMove) {
        boolean isValid = false;

        ChessPiece destPiece = this.aBoard.getPiece (aMove.getTarget());
        if ((destPiece == null) ||
            (!(destPiece.getPlayer().equals(this.aPlayer)))) {

                for (Behaviour aBehaviour : this.behaviours) {
                    if (aBehaviour.isValid (aMove, this.aBoard)) {
                        isValid = true;
                        break;
                    }
                }
        }

        return isValid;
    }
    ...
}
```

46

46

Pattern: Composite

Problem: <u>How do you manage a part-whole hierarchy of objects in a consistent way?</u>

Solution: *Define a common interface (or abstract base class) that both parts and composites implement (or inherit from).*

- Typically composite objects will implement their behavior by *delegating* to their parts.

47

47

Composite

Examples

- A Java GUI Container is a composite of GUI Components, and also extends `Component`.
- A `Group Element` contains multiple `Drawing Elements` (in fact, instances of concrete subclasses thereof)
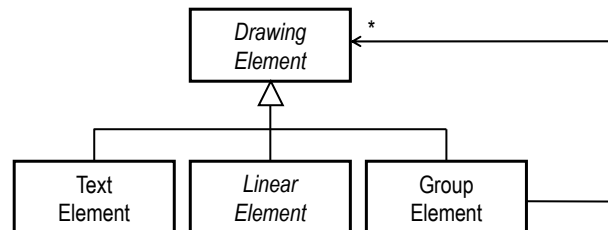
Consequences

- Clients can *uniformly manipulate* parts and wholes.
- In a complex hierarchy, *it may not be easy to define a common interface* that all classes should implement ...

48

48

Composite - Drawing Editor Example

49

Pattern: Observer

Problem: <u>How can an object inform arbitrary clients when it changes state?</u>
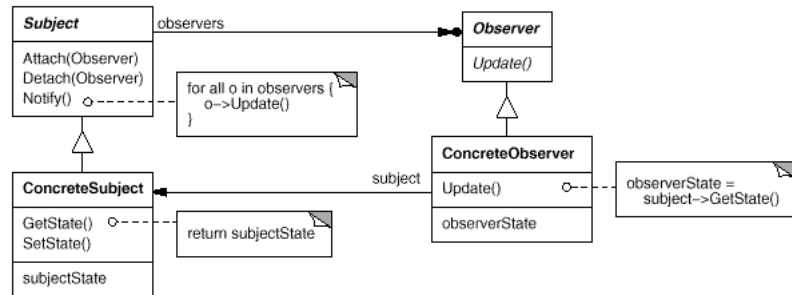
Solution: *Clients implement a common Observer interface and register with the "observable" object; the object notifies its observers when it changes state.*

- An observable object *publishes* state change events to its subscribers, who must implement a common interface for receiving *notification*.
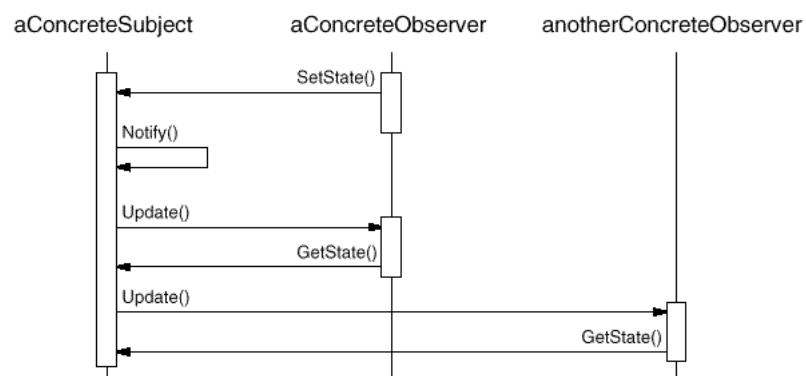
50

Observer — Structure

51

51

Observer — Collaborations

52

52

Observer

Examples

- A `Button` expects its observers to implement the `ActionListener` interface.

Consequences

- Notification can be *slow* if there are many observers for an observable, or if observers are themselves observable!

53

53

Pattern: Null Object

Problem: How do you avoid cluttering your code with tests for null object pointers?

Solution: *Introduce a Null Object that implements the interface you expect, but does nothing.*

- Null Objects may also be *Singleton objects*, since you never need more than one instance.

54

54

Null Object

Examples

- `NullOutputStream` extends `OutputStream` with an empty `write()` method

Consequences

- Simplifies client code

- Not worthwhile if there are only few and localized tests for null pointers

55

55

Pattern: Singleton

Problem: How do you ensure that a class has only a single instance and a well-defined access point to it?

Solution: *Introduce a Singleton class that keeps track of a static reference to its only instance, and provide a static method to access this sole instance.*

- Singletons are often used to give controlled access to specific resources (e.g., window manager, print spooler).

  ☞ `ChessBoard` is also a singleton as we will only ever have one board in a chess game.

56

56

Singleton - General Structure

```
public class Singleton {
  ...
  private static Singleton myInstance = null;
  protected Singleton() {...}
  ...
  public static Singleton getSingleton() {
    if (myInstance == null) {
      myInstance = new Singleton();
    }
    return myInstance;
  }
}
```

57

57

Pattern: Command

Problem: How do you let clients to be parameterized with different requests, queue or log requests, and support undoable operations?
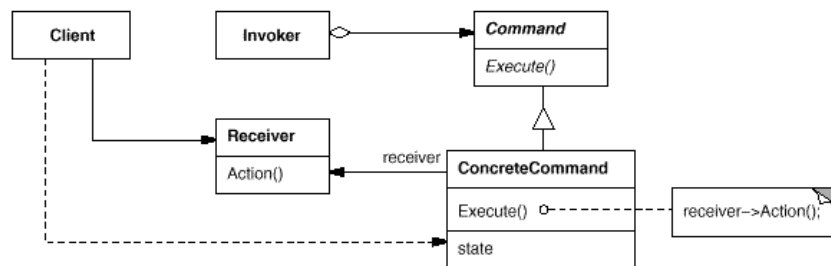
Solution: *Encapsulate requests in Command objects that can be passed around and executed where (and when) necessary.*

■ Command decouples an object that has to invoke an operation from the specific way how this operation has to be performed.

58

58

Command - General Structure

59

59

Command - Participants

Command

- declares an interface for executing an operation.

ConcreteCommand (PasteCommand, OpenCommand)

- defines a binding between a Receiver object and an action.
- implements Execute by invoking the corresponding operation(s) on Receiver.

Client

- creates a ConcreteCommand object and sets its receiver.

Invoker

- asks the command to carry out the request.

Receiver

- knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

60

60

Other Design Patterns

- Prototype:
  - □ Specify the kinds of objects to create using a *prototypical instance* and create new objects by copying this prototype.
- Proxy:
  - □ Provide a surrogate or *placeholder* for another object to control access to it.
- Iterator:
  - □ Provide a way to *access* the elements of an aggregate object *sequentially* without exposing its underlying representation.
- Visitor:
  - □ Represent an operation to be performed on the elements of an object structure without changing the classes of the elements on which it operates.
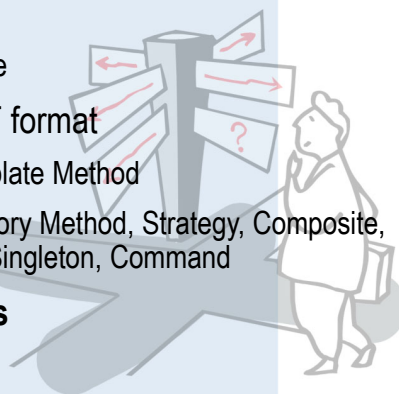
61

61

Outline

- **What are Patterns?**
- **Idioms**
  - □ Delegation and Interface
- **Design Patterns — GOF format**
  - □ Detailed example: Template Method
  - □ Common patterns: Factory Method, Strategy, Composite, Observer, Null Object, Singleton, Command
- **Other kinds of Patterns**
  - □ Pattern systems

62

62

Other Patterns

Patterns are not restricted to software designs, they can be found in many more problem domains:

- Reengineering Patterns
- Usability Patterns
- Documentation Patterns
- Organizational Patterns
- ...

Patterns have to be used with care: *"Do not overpattern!"*

63

63

Pattern for writing Abstracts

The *abstract* of a scientific paper gives an overview of its contents, the problem(s) it addresses, its contents, the approach taken, outcome, etc. Writing good abstracts is often an art. The following pattern can help:

1. Write one sentence about the main problem addressed in the paper.
2. Write one sentence why the problem stated is a real problem; motivate why dealing with the problem is of interest.
3. Summarize your approach to solve/address the problem in one sentence.
4. Finally, give evidence (one to two sentences) why your approach solves/addresses the problem.

☞ Exercise: write an abstract for "Inside Patterns" (recommended reading for this week).

64

64

Pattern Systems

■Patterns do not exist in isolation:

  □ they always work together with other patterns.

■A plain catalogue of patterns does not reflect all relationships:

  □ patterns should be interwoven in pattern systems.

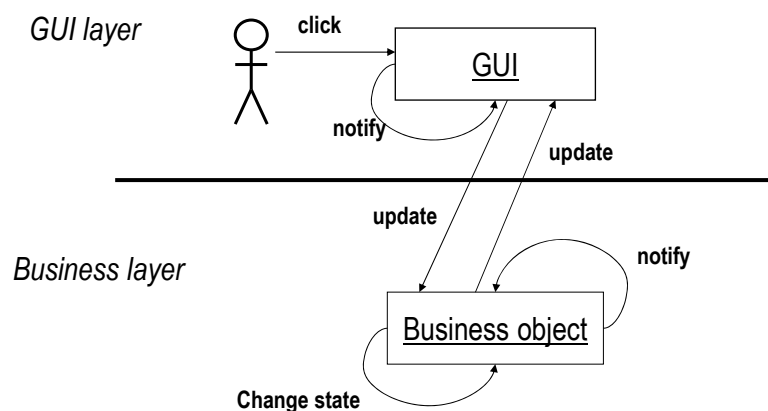■(Note: sometimes Pattern Systems are also referred to as *Pattern Languages*)

65

65

Observer + Layer

■ The *Observer* pattern is often used to keep objects in separate *Layers* independent

GUI layer

click

GUI

notify

update

update

Business layer

notify

Business object

Change state

66

66

Additional Readings about Patterns

- Christopher Alexander, et al., *A Pattern Language — Towns · Buildings · Construction*, Oxford University Press, 1977

- David Budgen, *Software Design (2nd Edition)*, chapter 10.

- Various articles in *Pattern Languages of Program Design* Series (published by Addison-Wesley)

67

67

Questions for Review

1. The author states that "A pattern provides a generic solution that can be implemented in multiple ways without ever being twice the same". Why can't software patterns be generalized to a single sample implementation or library for developers to use?

2. With regards to patterns in software development and their implementation, explain what *variations* refers to.

3. What is the difference between a "pattern" and a "framework" and how does this factor into how the software developer implements either of them?

4. What are some reasons that genericity is considered a good attribute of a Pattern?

68

68

Question to Answer – week 7 (for week 8)

The spec of the "Question to Answer" is under the corresponding assignment setup, which will be released this evening after the lecture.

69

69

Recommended Reading Week 8

■ Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice* (<u>Fourth</u> or <u>Third</u> Edition), Chapter 1 (available electronically through Swinburne Library)

OR

■ Len Bass, Paul Clements, Rick Kazman, *Software Architecture in Practice* (<u>Second</u> Edition), Chapter 2 (available electronically through Swinburne Library)

70

70