Design Patterns

The shift and focus (to patterns) will have a profound effect on the way we write programs

– Ward Cunningham & Ralph Johnson

On Design Patterns

- A **design pattern** systematically names, explains and evaluates an important and recurring design problem and its solution

- Good designers know not to solve every problem from first principles

  They reuse solutions

- This is very different from code reuse

- Software practitioners have not done a good job of recording experience in software design for others to use

Design Patterns – Definition

"We propose design patterns as a new mechanism for expressing object oriented design **experience**. Design patterns identify, name and abstract common themes in object oriented design. They capture the **intent** behind a design by identifying objects, collaborations and distribution of responsibilities."

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides ,"*Design Patterns*", Addison-Wesley, 1995. ISBN 0-201-63361-2

Others On Design Patterns

- Christopher Alexander

  "Each person describes a problem which occurs over and 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."

Others On Design Patterns – 2

- Cunningham

  “Patterns are the recurring solutions to the problem of design. People learn patterns by seeing them and recall them when need be without a lot of effort”

- Booch

  “A pattern is a solution to a problem in a specific context. A pattern codifies specific knowledge collected from experience in a domain.”

Patterns & Frameworks

- Patterns support reuse of software architecture and design

  They capture static and dynamic structures of successful solutions to problems. These problems arise when building applications in a particular domain

- Frameworks support reuse of detailed design and program source text

  A framework is an integrated set of components that collaborate to provide a reusable architecture for a family of related applications

Patterns & Frameworks – 2

- Frameworks tend to be less abstract than patterns

- Together, design patterns and frameworks help to improve key quality factors like reusability, extensibility and modularity

Classification – Creational

- Creational
  - » **Abstract the instantiation process**
    - > **Initializing and configuring classes and objects**
  - » **Make a system independent of how its objects are created, composed and represented**

- Class creational pattern uses inheritance to vary the class that is instantiated

- Object creational pattern delegates instantiation to another object

Creational Patterns

- Abstract Factory
  - » **Provide an interface for creating families of related or dependent objects without specifying their concrete classes**
- Builder
  - » **Separate the construction of a complex object from its representation so that the same construction process can create different representations**
- Factory Method
  - » **Define an interface for creating an object but lets subclasses decide the specific class to instantiate**

Creational Patterns – 2

- Prototype
  - Specify the kinds of objects to create using a prototypical instance and create new objects by copying the prototype

- Singleton
  - Ensure a class has only one instance and provide a global point of access

Classification – Structural

- Structural
  - » **Deals with how classes and objects are composed to form larger structures**
  - » **Decouple interface and implementation of classes and objects**
- Class structural patterns use inheritance to compose interfaces or implementations
- Object structural patterns describe ways to compose objects to realize new functionality

Structural Patterns

- Adapter
  - » **Convert the interface of a class into a different interface to let classes work together that otherwise could not**
- Bridge
  - » **Decouple an abstract from its implementation so that the two can vary independently**

Structural Patterns – 2

- Composite
  - » **Compose objects into tree structures representing part-whole hierarchies to deal uniformly with individual objects and hierarchies of objects**

- Decorator
  - » **Attach additional responsibilities to an object dynamically Provide a flexible alternative to sub-classing for extending functionality**

Structural Patterns – 3

- Façade
  - » **Provide common interface to a set of interfaces within system that defines a higher level interface to makes the system easier to use common tasks**
- Flyweight
  - » **Use sharing to support large numbers of fine-grained objects efficiently**
- Proxy
  - » **Provide a surrogate or placeholder for another object to control access to it**

Classification – Behavioural

- Concerned with algorithms and assignment of responsibilities among objects
  - » **Describe patterns of communication among objects**
  - » **Characterize complex run-time control flow**
- Class behavioural patterns use inheritance to distribute behaviour among classes
- Object behavioural patterns use object composition to distribute behaviour

Behavioural Patterns

- Chain of Responsibility
  - » **Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request by chaining the receiving objects and pass the request along the chain until an object handles it**
- Command
  - » **Encapsulate a request as an object thereby parameterizing clients with different requests, queue or log requests, and support undoable operations**
- Interpreter
  - » **Given a language, define a representation for tis grammar along with an interpreter that uses the representation to interpret sentences in the language**

Behavioural Patterns – 2

- Iterator
  - » **Access elements of a container sequentially without exposing the underlying representation**
- Master-Slave
  - » **Handles computation of replicated services in a system to achieve fault tolerance and robustness**
- Mediator
  - » **Define an object that encapsulates how a set of objects interact, promoting loose coupling by keeping objects from explicitly referring to each other and let them vary their interaction independently**

Behavioural Patterns – 3

- Memento
  - » **Without invalidating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later**
- Observer
  - » **Define one-to-many dependency, when one subject changes state, all observers (dependents) are notified and updated**
- State
  - » **Alter behaviour of an object when its internal state changes making the object appear to change its class**

Behavioural Patterns – 4

- Strategy
  - » **Define a family of algorithms, encapsulate each one, and make them interchangeable to let the algorithm vary independently from the clients that use it**
- Template Method
  - » **Define the skeleton of an algorithm in an operation, deferring some steps to subclasses while the structure of the algorithm does not change**
- Visitor
  - » **Represent an operation to be performed on all of the components of an object structure by defining new operations on a structure without changing the classes representing the components**

Acknowledgement

Descriptions of patterns

based on

Design Patterns

by

Erich Gamma, Richard Helm
Ralph Johnson, John Vlissides

Addison-Wesley, 1995.
ISBN 0-201-63361-2

Descriptive Template

- Name
- Intent

  What does the pattern do? What problems does it address?
- Motivation

  A scenario of pattern applicability
- Examples

  From real systems
- Abstract architecture

  General representation of the pattern
- Scenario – Collaborations

  How do the participants carry out their responsibilities?

Descriptive Template – 2

- Participants

  Describe participating classes/objects

- Applicability

  In which situations can this pattern be applied

- Consequences

  How does the pattern support its objectives?

- Implementation

  Pitfalls, language specific issues

- Related patterns

  Pointers to patterns dealing with similar problems and patterns used in conjunction with the current pattern

Becoming a Master Designer

- Learn the rules
  - » **algorithms and data structures**
  - » **languages**
  - » **mathematics**
- Learn the principles
  - » **structured and modular programming**
  - » **theory of software engineering**
  - » **OO design and programming**
- Study the designs of masters
  - » **Design patterns must be understood, memorized and applied**
  - » **Thousands of existing patterns**

    Are they all memorable?

Design Patterns Solve Design Problems

- Finding appropriate classes
- Determine class granularity

  How abstract, how correct

- Specify interfaces
- Specify implementation
- Put reuse to work

  Client vs inheritance

- Relate run time and compile time structures

  Program text may not reflect design

Design Patterns Solve Design Problems – 2

- Design for change is difficult
- Common problems
  - » **Explicit object creation**

    Use name of interface, not name of implementation
  - » **Dependence of particular operations**

    Avoid hard coded operations
  - » **Dependencies on hardware or software platforms**
  - » **Dependencies of object representation**
  - » **Dependencies on algorithms**
  - » **Tight coupling**

Claims of the Pattern Community

- Well defined design principles have a positive impact on software engineering
  - » **Achievable reusability**
  - » **Provide common vocabulary for designers**

    Communicate, document, explore alternatives
  - » **Patterns are like micro architectures**

    Useful for building small parts of a system
  - » **Reduce the learning time for understanding class libraries**
  - » **Avoid redesign stages by using encapsulated experience**

When to Use Patterns

- Solutions to problems that recur with variations
  - » **No need for pattern if the problem occurs in only one context**
  - » **Can we generalize the problem instance in which we are interested?**

- Solutions that require several steps
  - » **Not all problems need all steps**
  - » **Patterns can be overkill if solution is a simple linear set of interactions**

- Solutions where the solver is more interested in “does there exist a solution?” than in a solution’s complete derivation

  Patterns often leave out lots of detail

Key Principles

- Successful use of patterns and frameworks can be boiled down to a few key principles
  - » **Separate interface from implementation so each can vary independently**
  - » **Determine what is common and what is variable with an interface and an implementation**
  - » **Allow substitution of variable implementation via a common interface. Use deferred classes and effect them**
- Don't use blindly

  Separating commonalties from variabilities should be done on a goal by goal basis not exhaustively

  It isn't always worthwhile to apply them

Pattern Benefits

- Enable large scale reuse of software architectures
- Explicitly capture expert knowledge and design trade-offs
- Help improve developer communication
- Help ease the the transition to OO methods
- High level abstraction that leaves out the details

Pattern Drawbacks

- Patterns do not lead to direct code reuse
- Patterns are often deceptively simple
- You may suffer from pattern overload
- Patterns must be validated by experience and debate rather than automated testing
- Integrating patterns into a process is human intensive rather than a technical activity