Design Patterns

MSc in Communications Software

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

Design Patterns Selection

Identifying, Selecting and Using Design Patterns

Agenda

- Designing for Change
- Identifying Variability
- Using a Pattern

Design for Change

- The key to maintaining an agile application is to design a system that can evolve as requirements evolve.
    - Otherwise it risks major redesign costs in the future.
- Redesign might involve
    - class redefinition and re-implementation
    - client modification
    - retesting
- i.e. the “opening” of “closed” modules (Open Closed Principle)
- Design patterns help avoid this by ensuring that a system can change in specific ways.
    - Each design pattern lets some aspect of system structure vary independently of other aspects
    - Making a system more robust to a particular kind of change.

Common Causes of Redesign

1. Creating an object by specifying a class explicitly
2. Dependence on specific request/operation fulfillment mechanisms
3. Dependence on hardware and software platform
4. Dependence on object representations or implementations
5. Algorithmic dependencies
6. Tight coupling
7. Extending functionality by implementation inheritance
8. Inability to alter classes conveniently

(1) Creating an object by specifying a class explicitly

- Specifying a class name when creating an object commits code to a particular implementation (class) instead of a particular type (interface)
- This commitment can complicate future changes.
- To avoid it, create objects indirectly.

Abstract Factory
Factory Method
Prototype

(2) Dependence on specific request/operation fulfillment mechanisms

- When specifying a particular operation (particular user originated) the design can commit to one way of satisfying a request.
- By avoiding hard-coded operations the design can make it easier to change the way a request is fulfilled

Chain of Responsibility
Command

(3) Dependence on hardware and software platform

- External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms.
- Software that depends on a particular platform will be harder to port to other platforms.
- It may even be difficult to keep it up to date on its native platform.
- Design system to limit its platform dependencies.

Abstract Factory
Bridge

(4) Dependence on object representations or implementations

- Clients that know how an object is represented, stored, located or implemented might need to be changed when the object changes.
- Hiding this information from clients keeps changes from cascading.

Abstract Factory
Bridge
Memento
Proxy

(5) Algorithmic dependencies

- Algorithms are often extended, optimized, and replaced during development and reuse.
- Objects that depend on an algorithm will have to change when the algorithm changes.
- Therefore algorithms that are likely to change should be isolated.

Builder
Iterator
Strategy
Template Method
Visitor

(6) Tight coupling

- Tightly coupled classes are hard to reuse in isolation, since they depend on each other.
  - Produces monolithic systems whereby classes cannot be changed or removed without understanding and changing many other classes.
  - The system becomes a dense mass that's hard to learn, port, and maintain.
- Loose coupling increases the probability a class can be reused
  - Produces a system can be learned, ported, modified, and extended more easily.

Abstract Factory
Bridge
Chain of Responsibility
Command
Facade
Mediator
Observer

(7) Extending functionality by subclassing

- Complexities with implementation inheritance:
  - Implementation overhead (initialization, finalization, etc.).
  - In-depth understanding of the parent class.
  - Overriding one operation might require overriding another.
  - An overridden operation might be required to call an inherited operation.
- Object composition & delegation provide flexible alternatives:
  - Functionality can be introduced by composing existing objects in new ways
  - Many patterns facilitate introduction of new functionality just by implementing an interface & composing its instances with existing objects.

Bridge
Chain of Responsibility
Composite
Decorator
Observer
Strategy

(8) Inability to alter classes

- Sometimes a class requires modification, but:
  - Source code is unavailable
  - Change would require modifying many existing subclasses.
- Design patterns offer ways to modify behaviour of such classes in some circumstances.

Adapter

Decorator

Visitor

Identifying Variability

- Consider carefully what should be variable in the design
- Instead of considering
  - what might force a change to a design
- consider
  - what you want to be able to change without redesign.
- The focus is on encapsulating the concept that varies, a theme of many design patterns.

Purpose	Design Pattern	Aspect(s) That Can Vary
Behavioral	Chain of Responsibility	object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed, traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	states of an object
	Strategy	an algorithm
	Template Method	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Purpose	Design Pattern	Aspect(s) That Can Vary
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	responsibilities of an object without subclassing
	Façade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	how an object is accessed; its location

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory	families of product objects
	Builder	how a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	the sole instance of a class

Patterns Not a Panacea

- Design patterns should not be applied indiscriminately.
- Often they achieve flexibility and variability by introducing additional levels of indirection
    - Complicating a design
    - Degrading performance.
- A design pattern should only be applied when the flexibility it affords is actually needed.

Summary Guidelines (1)

- Read the pattern documentation through for an overview.
  - Particularly Applicability and Consequences sections (if available)
- Study the Structure, Participants, and Collaborations sections.
  - Be sure to understand the classes and objects in the pattern and how they relate to one another.
- Study the sample code carefully
- Choose names for pattern participants that are meaningful in the application context:
  - The names for participants in the patterns may be too abstract to appear directly in an application. Nevertheless, it may be useful to incorporate the participant name into the name that appears in the application.

Summary Guidelines (2)

- Define the classes:
  - Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references.
  - Identify existing classes in the application that the pattern will affect, and modify them accordingly.
- Define application-specific names for operations in the pattern.
  - Again the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide.
- Implement the operations to carry out the responsibilities and collaborations in the pattern.
  - The Implementation section in the pattern documentation may offer hints to guide the implementation.

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit