CH: 0 Patterns: A solution to a problem in a context: A three part rule: a problem, a system of forces, and a solution to balance the forces. Coplien Form. Name: Crucial because it is what the pattern is remember by, should be memorable. Context: Describes where the pattern is useful. Generally before the pattern is applied. Be specific. Problem: Describes something wrong, a problem statement. Forces: Describes what makes the problem hard, why won’t the obvious solution work. Almost always multiple, that conflict. Solution: First part written, usually not obvious, one or two sentences that captures the essence, followed by paragraphs that give details, typically emergent. Sketch: visual representation, usually structural. Resulting Context: How has the world changed by the pattern, what problems are created or solved. Ch: 1 Design Patterns: Nouns: Classes/Objects, Responsibilities: Methods, Inheritance: Is-a relationship. Design is a concept and inheritance has a benefit of code sharing, but should be established statically, before runtime. Polymorphism separates what changes polymorphism via virtual functions separates what changes (the implementation) from what stays the same (the interface). Implementations go in their own hierarchies. Allows for low coupling, allows changes at runtime. Strategy Pattern. Benefits of Patterns: easily used/interchangeable. Design chunks at a higher level. Patterns short-circuit the *discovery interval* for many important design structures. More importantly, patterns avoid rework that comes from inexpert design decisions. Programmers who don’t understand [patterns] … spend a long time converging on a solution, or will employ solutions that are less maintainable or just plain wrong. Patterns Guide Humans Ch 2: Observer a GUI window contains *views* of some data. The same data can have multiple *views* in use simultaneously The GUI also has *control* mechanisms (buttons, etc.) The data should have *no knowledge* of its viewers (or that it even has any viewers!) But the views must somehow be updated as the data changes and/or as user events occur. Our data will have minimal knowledge of observers (through an interface only) A subject maintains a list of observers. All observers are notified (when, composite updates, threading?) Observers keep a reference to the subject, so they can query state. Notify method calls update for each observer, what about passing data? Pull: subject notifies w/o sending data. Observers then call getter methods. Observers must know the interface for the subject, have to figure out what changed. Push: Subject passes data in a parameter in update. Observers must know how to interpret the data. Observers are less reusable. Threads may compete for same resource. NotifyAll is an application of observer pattern. How to implement variations: already existing class (wrap it). Polling, taking a constant poll of update status. A mediator can do the polling as well. Proxy class will stand in for the subject and implement the interface, and notify observers when the subject changes. Definition: A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context, in this class form we use: Name Problem, Intent, Context, and Solution. Choose a pattern by intent (the goal), not structure. Intent: Describes the goal in succinct, high-level terms. Context: Describes the problem, including the forces that cause the “dilemma”, and the negative consequences of not changing the design. Solution: Describes the redesign that resolves the forces in a positive way. Include a sketch. Ch. 3. Combinatorial Explosion 2^n. (inheritance web) be broader. It’s better to extend a class but y not changing its code. If we change code, clients that depend on it will have to change, new bugs emerge, inheritance web. Wrap classes to extend them. The wrapped functions call the inner functions. Decorator can query what it surrounds, and can access other types of decorators, not quite programming to an interface. Favors composition over inheritance. Ch 4. How to ensure that the correct subclass gets created? What if other things have to happen in conjunction with creation of an object? What if you have to create two or more related objects? Ultimately, the exact object of a type must be known in order to create it (duh) Foo f; Bar b = new Bar(“baz”);But this violates an important principle: Program to an interface, not an implementation Or, program to *abstractions*, not *concrete types*Concrete object creation is best localized in *one place* So introducing new types doesn’t cause maintenance trauma. Static (Single class) factory. Provide a static method that returns a point to a heap object. Make all constructors not public. No variable in an abstraction should hold a pointer to a concrete class No class should derive from a concrete class No method should override an *implemented* method of any of its base classesOnly override abstract methods These rules can’t be followed all the time! Emphasis on *thumb* :-) The key is how *volatile* the lower-level module is

Principles: 1. identify what changes and separate it from what stays the same: underlies most patterns. 2. Program to an interface, not an implementation. 3. Do not repeat yourself. 4. Favor composition over inheritance (composition is more flexible) 5. Strive for loosely-coupled designs between objects that interact. 6. Open-Closed Principle: classes should be open for extension but not modification. 7. Resource Acquisition is Initialization (RAII), whenever a resource is acquired, make sure it is initialized at the same time. 8. Dependency inversion principle, high-level modules should not depend on low-level modules

Patterns: 1. Strategy (policy), define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets each algorithm vary independently from clients that use it. Disadvantages: Lots of extra classes (who cares?), other clients may need to know about the strategy, increased overhead. 2. Observer Pattern: Allows an object to be tracked (observed) by an arbitrary number of observers with a minimum of coupling. Advantages: Both sides are insulated from implementation details. Concrete implementations may vary. New observers can be added at will. Subject doesn’t change when new types of observers are added. 3. Decorator Pattern (structural) Intent: Add additional responsibilities to an object *dynamically*. Context: Applies when clients program to an abstraction (not its subtypes). A decorator can be used in the same way as its subject (“is-a”). The set of additional responsibilities can be open-ended. Existing subject code does not change. Multiple decorators can be combined sequentially. Solution: The Decorator implements the same interface as the abstraction. It manages a concrete instance of the abstraction via composition, and adds additional functionality. It calls the managed object’s methods and combines the result with the additional functionality. The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub classing for extending functionality

