**Principles: Role of Inheritance:** IS-A relationships - make subtypes. - side benefit of code sharing -design is best benefit -don't overuse, stated statically, too rigid, not everything should be inherited **Don't repeat yourself:** Repeat Bugs, Code Bloat, Repeating changes. Extra time - worth it? Small Segments make duplication identical, increases coupling of duplication is in unrelated modules. Functions, Objects, Inheritance, Patterns: Strategy, Null Object, Template Method State (Most patterns) **Identify the aspects of your application that vary and separate them from what stays the same:** humans do this. Underlies most patterns. Code is harder to understand, files modified for other reasons, changes affect users incidentally. How: Observer, Command, Template Method, Null Object **All problems can be solved by adding another level of indirection** -- proverb. Create a new class, encapsulate what is needed, then attach it. **Pointer to implementation (PIMPL):** one way of implementing the proverb. **Program to an interface not an implementation:**  implementation may change, so encapsulate what varies and add it on. How: Inheritance, PIMPL, Adaptor, Facade, Proxy **Favor Composition of Inheritance:**  composition (has-a, uses-a) is more flexible resist the urge to over-classify hierarchies. adds flexibility at runtime, easier to separate interface from implementation, easier to maintain. Increases cohesion, How: strategy **Single Responsibility Principle (SRP):** A class or function has only one job, only one reason to change, not just one method. bad: large classes /methods / functions, many conditionals (complex code), unrelated conditionals, increase maintenance, large parameter list(Symptom) How: strategy, state, facade, iterator, split classes / functions / methods **Open-Closed Principle:**  open for extension closed for modification. Better in extend a class by not changing its code. Clients dependent on it will need to change, new bugs emerge, inheritance violates this principle. Compose a wrapper to extend a class. Consequences: change the base class, harder to maintain, changes cause clients to change. How: Decorator, adapter, redesign hierarchy, Dependency Inversion Principle, Separate interface from implementation. **Law of Demeter**: Don't talk to strangers, avoids transitive coupling, each object must provide a complete interface, can't delegate parts of its interface to objects it contains, makes fat interfaces. How to reduce: Facade, maybe adapter **Liskov Substitution Principle:** An instance of a derived class can be used wherever an instance of the base class is used. Base class can define interface and maybe behavior. Derived classes must fulfill the interface, and may extend the behavior. Not recommended to override base class behavior (negative polymorphism). How: Decorator, template method, composite, proxy a general OO design approach. The derived class's implementation of the interface must honor the intent. **Dependency Inversion Principle:**  concrete items may depend on abstract items but abstract items should not depend on concrete items. A base class should not depend on any of its derived classes. bad: casts down the inheritance hierarchy, checks for object type, avoid if you can. How: Decorator **Resource Allocation is Initialization:**  Once you create a resource allocate it, How Factories, (possibly singleton). Constructors help **Interface Segregation Principle:** clients should not be forced to depend on methods they do not use, How: adaptor, Facade

**Code Smells:** principle violation warnings: **- Duplicate Code** **- Long Method** **- Large class**, too many data members, or methods, lots of duplicate code, cohesion problems **-Long parameter list**, hard to understand easy to get wrong **-Divergent change**, changes for different reasons, pulls in different directions. (cohesion) **Shotgun Surgery,** opposite of divergent change, one change causes many small changes across multiple classes (coupling). **-Data Clumps**  groups of data that like to hang out together (WHY?). **-Primitive Obsession,** lots of primitive types. What's the benefit of the object? -**Switch Statements -Parallel Inheritance Hierarchies**

Patterns: a problem, a system of forces, and a solution that balances the forces: Name, context, problem, forces, solution, sketch, resulting context. (+) identify solutions and implementations, think at higher level, design chunks, short-circuit the discovery interval, avoid rework **-Strategy:** (behavioral) Encapsulates interchangeable behaviors and uses delegation to decide which one to use, Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. (+) Reduces coupling, invoke at runtime / dynamic allows us to change behavior (-) extra classes, subclasses need to be aware of different strategies, increased overhead due to indirection (can't make inline) **-Facade:** (structural)Simplifies the interface of a set of classes, delegates client requests to appropriate subsystem class, Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. **-Decorator:** (structural)Wraps and object to provide new behavior, Define an interface for creating an object, but let subclasses decide which class to instantiate. (+) encapsulation, composition over implementation, (-) undo is hard and ugly **- Factory Method**: (creational) lets a class defer instantiation to subclasses. Subclasses decide which concrete classes to create, Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (+) decoupling, separate object creation from object use, follow DIP **-Template Method:** (behavioral) Subclasses decide how to implement steps in an algorithm, 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. (+) Template Method is in charge, reduced duplicate code, form of delegation **-Abstract Factory:** (creational)Allows a client to create families of objects without specifying their concrete classes, Provide an interface for creating families of related or dependent objects without specifying their concrete classes. **-State:** (behavioral)Encapsulates state-based behaviors and uses delegation to switch between behaviors, Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. State classes have a link back to the context, so they can change the state and call other context methods. **-Adaptor:** (structural)Wraps and object and provides a different interface to it, Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. (+) allows different classes / object to communicate / changes interface (Facade simplifies, adapter converts) -**Composite:** (structural) clients treat collections of objects and individual objects uniformly, Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. **-Command:** (behavioral) Encapsulates a request as an object, Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. (+) decoupling, Macros, Undo, Batch processing, save a log / roll back Dynamic command creation Separation of command creation from command execution Encapsulated sequences of operations That could be reused or used by different objects Ability to schedule operations Ability to apply the command to different “command-ees”, or receivers Ability to undo **-Iterator:** (behavioral) Provides a way to traverse a collection of objects without exposing its implementation, Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. Reification of a traversal of a collection, SRP, 1. a particular object, 2. knows current element, 3 next element, 4. knows the end. **-Proxy:** (structural) Wraps and object to control access to it, Provide a surrogate or placeholder for another object to control access to it. Used to overcome barriers. **-Observer:** (behavioral) Allows objects to be notified when state changes, Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. (+) both sides insulated from implementation details, add new observers, subject doesn't change when new types are created, subject maintains list of observers, all observers are notified, observers keep reference to subject to they can query the state, (push vs pull)

**-Singleton:** (creational)Ensures one and only object is created, Ensure a class has only one instance and provide a global point of access to it. done by static constructors. (+) lazy instantiation (-) just a global variable?, repeated if statements, need to destroy it

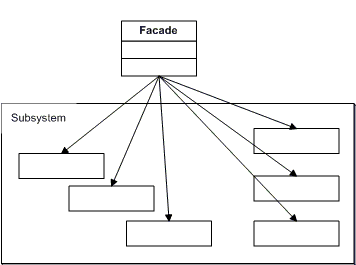
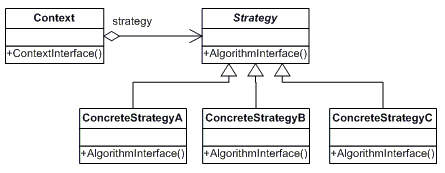
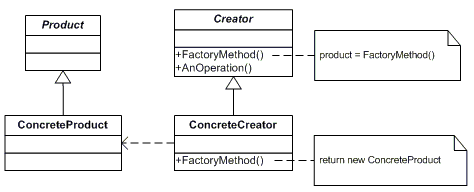
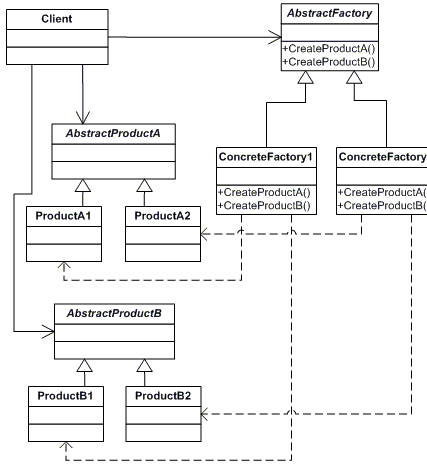
**-Null Object:** something for nothing, applies when you always check before invocation, default to do nothing or some constant action,

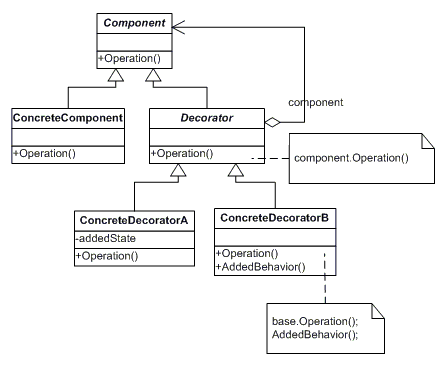
**Resource MGMT Patterns:**

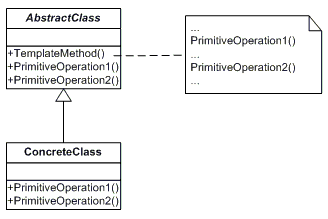
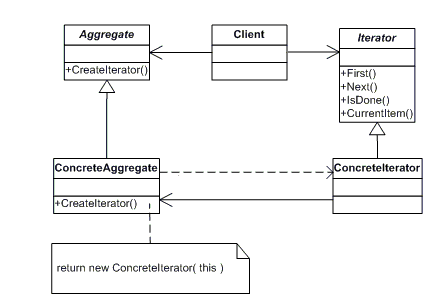
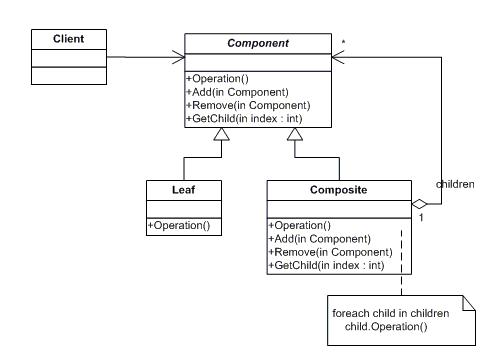
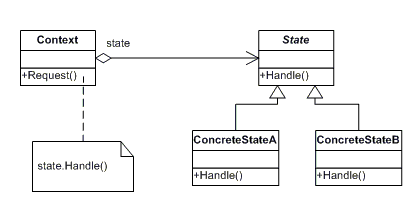
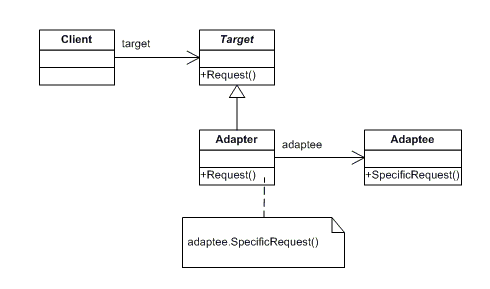
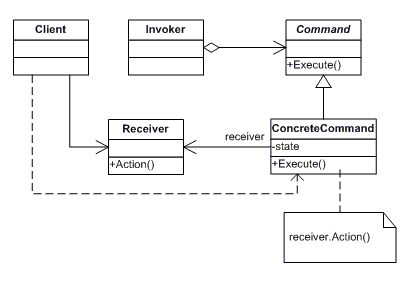
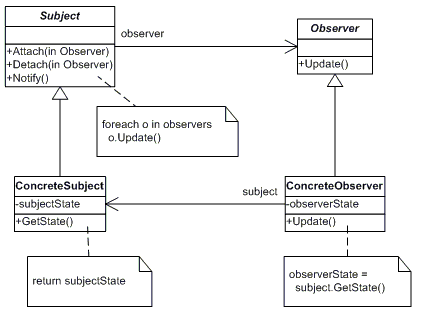
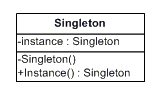
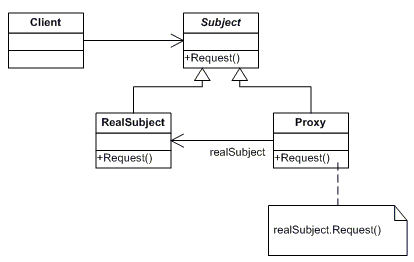
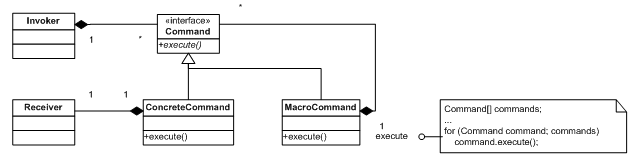
**-Creator as sole owner:** function as owner, object as owner, class as owner **-Sequence of owners: -Shared Ownership:**

**Patterns of Robust software:**

**-Object Pool:** manage a pool of objects with defined behavior (graceful failure), checksum, dangling pointer  **-Leaky Bucket Counter:** Ignore small errors handle large frequent ones, **-Resource Audits:** periodically check to make sure at a correct state, do in background / leisure time **-Tend to Stable State:** when an error is encountered go to a stable state.  **-watchdog:** checks for infinite loops: sends ack and waits for response otherwise kills process. Who watches the watchdog. **-Escalating Restarts:** Define levels of restart, each more serious than the last, re-initialize, restart a process, restart all process, restart OS, power down and up.  **-Half Object Plus Protocol: -Overload handling:** Finish what you start, Shed Work Periphery: refuse to handle work, Fresh work before Stale, Ignore Babbling Idiots **-Failover**  Load Sharing (N+ M sparing), Cold, warm, hot Standby, **-Upgrades** Rolling, Hitless

****

****

****