Progettare Per Preservare

- Creare un oggetto specificandone la classe esplicitamente **orienta ad una particolare implementazione invece che a una interfaccia**, questo può complicare i cambiamenti futuri
- E' meglio creare oggetti indirettamente
- Pattern utili: **Factory Method**, Prototype

- Le **dipendenze da operazioni specifiche vincolano ad un modo di soddisfare una richiesta**, evitando che le richieste siano indicate nel codice permette di modificare a compile-time e a run-time il modo in cui le richieste saranno soddisfatte
- Pattern: Chain of Responsibility, Command

Progettare Per Preservare

- **Dipendenze da piattaforme hardware e software**, ovvero API esterne specifiche di una piattaforma, rendono più difficile fare il porting del sistema software verso altre piattaforme. E' persino difficile tenerlo aggiornato per la stessa piattaforma.
- Progettare il sistema limitando le dipendenze dalla piattaforma
- Pattern: Factory, Bridge
- **Dipendenze da rappresentazioni o implementazioni** (come un oggetto è rappresentato, conservato, o implementato) rendono le classi client soggette a cambiamenti quando l'oggetto cambia
- Nascondere le informazioni alle classi client evita cambiamenti a cascata
- Pattern: Factory, Bridge, Memento, Proxy

Progettare Per Preservare

- **Dipendenze di oggetti da algoritmi** che sono spesso estesi, ottimizzati e rimpiazzati durante lo sviluppo ed il riuso, provocano cambiamenti agli oggetti nel caso di cambiamento degli algoritmi
- Si devono isolare gli algoritmi soggetti a cambiamenti
- Pattern: Builder, Iterator, Strategy/State, Template Method, Visitor

- **Lo stretto accoppiamento fra varie classi** porta a sistemi monolitici. **Classi strettamente accoppiate** sono difficili da modificare, e riusare singolarmente, senza comprendere e cambiare le altre classi
- Il lasco accoppiamento incrementa la probabilità che una classe sia riusata da sola, e che il sistema sia compreso, modificato, ed esteso più facilmente
- Pattern: Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator, Observer

Progettare Per Preservare

- **L'estesione di funzionalità tramite sottoclassi** è riuso a scatola bianca (white box reuse), e richiede profonda comprensione della superclasse e l'override di un'operazione può richiedere l'override di un'altra. Può condurre a un grande numero di classi (per estensioni)
- E' meglio **limitare il numero di livelli in una gerarchia** di classi e usare la composizione anziché l'ereditarietà: black box reuse
- Alcuni design pattern producono una progettazione in cui si possono personalizzare le funzionalità definendo solo una sottoclasse e componendo le sue istanze con le altre
- Pattern: Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Quando è **impossibile modificare classi**, per es. poiché il codice sorgente non è disponibile, o quando una modifica può richiedere cambiamenti a tante altre classi
- Pattern: Adapter, Decorator, Visitor