

# Antipattern

- Antipattern
  - Sono soluzioni di progettazione ed implementazione che si sono rivelate inefficaci
  - Sono descritti da
    - Nome dell'antipattern
    - Forma: situazione che ha conseguenze negative
    - Sintomi: come riconoscere l'antipattern
    - Cause: cosa porta a produrre l'antipattern
    - Soluzione: come eliminare l'antipattern

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 1

# Antipattern God class (aka Blob)

- Forma
  - Una classe di grandi dimensioni, la God class o Controller, usa tante classi di piccole dimensioni, le quali spesso contengono solo dati
  - Controller ha tante responsabilità, è difficile da comprendere
  - Controller invoca molti metodi di altre classi, Controller non contiene i dati su cui lavora
  - L'architettura del sistema che usa la God class è di stile procedurale
- Sintomi
  - Una classe con tanti attributi e metodi (60 o più)
  - Tale classe manca di coesione (poiché ha tante responsabilità)
  - Tale classe è troppo complessa per essere riusata o testata
- Cause
  - Mancanza di una architettura ad oggetti, i progettisti non hanno adeguata comprensione dei principi OO
  - Sono state aggiunte funzionalità al sistema senza revisionare l'allocazione di responsabilità

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 2

# God class

- Soluzione
  - Refactoring: distribuire le responsabilità tra varie classi
  - Una classe dovrebbe contenere i dati che permettono ad essa di prendere delle decisioni
  - Identificare set di dati ed operazioni coesi tra loro e creare una nuova classe per ogni set identificato
  - Rimuovere interazioni tra classi che hanno relazioni "logiche" indirette

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 3

# Lava Flow

- Forma
  - In un sistema software, nato da una ricerca, ci sono parti di codice irrimovibili (lava solidificata) che nessuno conosce
  - Tali parti sono risalenti ad uno sviluppo precedente, quando gli sviluppatori, in fase di ricerca, hanno provato varie soluzioni per risolvere un problema
  - Sono presenti frammenti di codice non chiaramente relazionati con il resto del sistema
  - Il codice Lava Flow è difficile da analizzare, verificare, testare, e spreca risorse di memoria e di processore
- Sintomi
  - Parti di codice e variabili non spiegabili, non documentate, non relazionate al resto
  - Parti di codice commentato senza spiegazioni
  - Codice non usato, poiché mai richiamato da altre parti (dead code)

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 4

# Lava Flow

- Cause
  - Codice di ricerca è diventato codice di produzione
  - E' stato distribuito codice non completo
  - Mancanza di architettura
  - Architettura cambiata in fase di implementazione
- Soluzione
  - Assicurarsi che una architettura software appropriata guidi lo sviluppo
  - Per definire l'architettura è necessario scoprire quali sono le attività del sistema esistente e localizzare i componenti realmente usati e utili
  - Scoprire le attività del sistema permette di identificare linee di codice inutile
  - L'eliminazione del codice inutile provoca l'inserimento di difetti che possono essere corretti solo comprendendo bene la causa dell'errore
  - Per evitare Lava Flow identificare e documentare interfacce software stabili

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 5

# Functional Decomposition (aka No OO)

- Forma
  - Risultato del lavoro di sviluppatori con esperienza non OO che progettano ed implementano un'applicazione in un linguaggio OO
  - Il codice assomiglia ad un linguaggio strutturale (Pascal, C, Fortran)
- Sintomi
  - Classi con nomi di funzione, es. CalcolaInteresse
  - Classi con una singola operazione (o poche operazioni)
  - Nessun uso di ereditarietà e polimorfismo
  - Le classi presenti sono difficili da documentare, poiché senza senso
  - Sistema difficile da testare
- Cause
  - Sviluppatori che non hanno compreso la tecnologia OO
  - Mancanza di una architettura ben definita

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 6

# Functional Decomposition

- Soluzione
  - Analizzare il sistema software e determinare quali sono i requisiti principali
  - Documentare tali requisiti
  - Produrre un design che incorpora le parti principali del sistema
  - Combinare insieme piccole classi che insieme hanno un unico obiettivo
  - Se una classe non contiene uno stato riscriverla come una funzione

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 7

# Spaghetti Code

- Forma
  - Le progressive estensioni hanno compromesso la struttura a tal punto che non è più comprensibile, la struttura sembra inesistente
  - Il sistema è difficile da mantenere ed estendere
- Sintomi
  - I metodi sono lunghi, senza parametri e usano variabili globali
  - Il flusso di esecuzione è obbligato dall'implementazione degli oggetti, non dai client degli oggetti
  - Le interazioni tra oggetti sono minime
  - Nomi di classi e metodi suggeriscono la programmazione procedurale
  - Ereditarietà e polimorfismo non sono usati, riuso impossibile
  - Gli oggetti non mantengono stato tra varie invocazioni di metodo
- Cause
  - Inesperienza con la tecnologia OO
  - Nessuna progettazione prima dell'implementazione

Prof. E. Tramontana - Anti Pattern - 14-Giu-10 8

# Spaghetti Code

- Soluzione
  - Convertire frammenti di codice in metodi
  - Rimuovere codice che possono diventare inaccessibili
  - Rinominare classi, metodi e dati per rendere il codice più facile da leggere
  - Condurre una progettazione orientata agli oggetti
  - Identificare oggetti piccoli, comprensibili agli sviluppatori

# Cut-and-paste Programming

- Forma
  - Frammenti di codice simili sparsi nel sistema, duplicazione di codice
  - Programmatori con poca esperienza che prendono esempio da programmatori con esperienza
  - Modifiche a breve termine che soddisfano i requisiti
- Sintomi
  - Lo stesso bug si ripete malgrado molte correzioni sono state effettuate
  - Linee di codice vengono aggiunte senza migliorare la produttività
  - Difficile localizzare e sistemare tutte le istanze di un certo errore
  - Alti costi di manutenzione, difetti replicati
  - Tante correzioni ma nessun modo di correggere le tante versioni
  - Tipo di riuso che falsamente incrementa il numero di linee di codice

# Cut-and-paste Programming

- Cause
  - Creare codice riusabile è faticoso e si preferisce ritorno d'investimento immediato anziché a lungo termine
  - La velocità di sviluppo del codice è più importante dei fattori di qualità
  - Gli sviluppatori non lavorano con le astrazioni, non hanno compreso ereditarietà e composizione
  - I componenti una volta creati non sono documentati sufficientemente e resi disponibili prontamente agli altri sviluppatori
  - Gli sviluppatori non pensano a quello che verrà dopo
- Soluzione
  - Ridurre o eliminare cloning
  - Fare refactoring per eliminare le tante versioni tramite parametrizzazione, e creare un set di componenti riusabili
  - Riusare tali componenti in modalità black-box, ovvero tramite composizione

# Excessive Dynamic Allocation

- Problema
  - Creazione e distruzione frequente di oggetti di una stessa classe
    - Decadimento delle prestazioni, se avviene su un grande numero di oggetti
    - Per la creazione: allocazione di memoria, inizializ. codice, inizializ. oggetto
    - Per la distruzione: esecuzione del garbage collector
- Soluzione
  - Cambiare il codice per eliminare molte esecuzioni di new
    - Ad es. se new è dentro un ciclo, potrebbe essere portato fuori
  - Riciclare oggetti anziché crearne di nuovi (gestione con object pool)
  - Eliminare la necessità di avere nuovi oggetti (usando ad es. il pattern Flyweight)
    - Individuare e condividere lo stato intrinseco immutabile (oggetto Flyweight) e separarlo dallo stato estrinseco (dipendente dal contesto)