UML

UML é una linguaxe gráfica para visualizar, especificar, construir e documentar. Proporciona una forma estándar de escribir os planos dun sistema.

Un **modelo** é una simplificación da realidade, completa e consistente. Os bos modelos resaltan elementos de influencia e ocultan os non relevantes para o **nivel de abstracción** dado. Poden ser **estruturais** (organización) ou **de comportamento** (dinámica).

Modélase para producir software de calidade duradeira que satisfaga de xeito consistente e predecible necesidades cambiantes, cumpra o seu propósito e sexa rápido e efectivo. Hai 2 enfoques: algorítmico (tradicional) e orientado a obxectos (moderno).

Principios de modelado:

- A elección dos **modelos** inflúe en como se acomete un problema e a súa solución.
- Todo modelo se pode expresar en distintos niveis de **abstracción**.
- Os mellores modelos están ligados á **realidade**.
- Non abonda cun só modelo.

Os bloques de construción de UML son **elementos**, **relacións** e **diagramas**.

Un **diagrama** é un grafo conexo de nodos e arcos. Un diagrama ben estruturado céntrase nun só aspecto do sistema, contén só elementos e relacións esenciais para el, e proporciona detalles consistentes co seu nivel de abstracción.

Diagrama de casos de uso

Os **casos de uso** capturan o comportamento desexado do sistema sen especificar como implementalo. Son conxuntos de secuencias de accións que executa un sistema para producir un resultado observable de valor para un actor.

Unha **secuencia** é unha interacción do sistema con elementos externos. Un **actor** é un rol que xoga un usuario ou sistema ao interaccionar co sistema. O mesmo actor pode desenvolver varios roles. Un **escenario** é unha instancia do caso de uso.

Os **casos de uso** especifícanse textualmente como un fluxo de eventos, incluíndo un **escenario principal** e as súas **alternativas**. Capturan comportamentos sen especificar a implementación, e realízanse creando **colaboracións** de clases e outros elementos. Un caso de uso ben estruturado denota un comportamento simple e identificable, identifica **actores** que interactúan con el, incorpora comportamento común incluíndo outros casos de uso e coloca variantes en casos de uso que o estenden, describe o fluxo de eventos con usuarios por medio de escenarios, e especifica **pre** e **post condicións**.

Os **diagramas de casos de uso** visualizan o **comportamento** dun sistema e empréganse para modelar o **contexto** (que elementos están fóra e cales dentro) e os **requisitos funcionais** (especificar que debería facer)dun sistema. Componse de:

- **Actores**
- **Casos de uso**
- **Asociacións** entre actores e casos de uso
- **Relacións** entre casos de uso (xeralización, inclusión, extensión) ou entre actores (xeralización).

Na **xeralización** hérdase o comportamento dun caso de uso, podendo engadir ou redefinir comportamento; para modelar casos de uso que fan un pouco máis. Na **inclusión** un caso de uso

incorpora o comportamento doutro; para evitar repetir comportamento común. Na **extensión** un caso de uso modifica o comportamento doutro en **puntos de extensión**; para describir variacións.

Non deben mostrar estrutura, fluxo ou despregue.

Diagrama de clases

Unha **clase** é unha descrición dun conxunto de obxectos que comparten atributos, operacións, relacións e semántica. Son **abstraccións** independentes da linguaxe de programación capaces de implementar **interfaces**. Os obxectos da mesma clase teñen os mesmos tipos de **estado** e **comportamento**. As clases **colaboran** entre si para satisfacer necesidades. Hai 3 tipos de **relacións**: **dependencia** (os cambios na especificación dun elemento poden afectar a outro, indica utilización), **xeralización** (relación pai/fillo) e **asociación** (relación estrutural). Son a base para diagramas de paquetes, compoñentes e despregue. Poden conter **paquetes**.

Diagrama de interacción

Unha **interacción** é un comportamento que inclúe **mensaxes** intercambiados por un conxunto de obxectos dentro dun **contexto** para lograr un propósito. Modela dinámica de **colaboracións** (sociedade de obxectos que proporcionan un comportamento maior que a suma dos seus comportamentos). Unha **mensaxe** é unha transmisión de información entre obxectos para desencadear unha actividade. Poden ser: **chamadas** (invocan operacións), **retornos** (devolven un valor), **sinais**, **creación** ou **destrución**. As interaccións serven para modelar **fluxo de control** nun sistema ou subsistema, na implementación dunha operación, ou nunha clase ou compoñente. Os obxectos poden ser **concretos** ou **prototípicos**. Os **enlaces** son conexións semánticas entre obxectos polas que se transmite unha mensaxe. Unha **secuencia** é un fluxo de mensaxes encadeadas entre diferentes obxectos, orixínase dentro dun **proceso** ou **fío** e continúa mentres este exista; cada proceso redefine un **fluxo de control** separado no que as mensaxes se ordenan segundo se suceden no tempo.

Os **diagramas de interaccións** describen a forma na que **colaboran** distintos obxectos para producir un comportamento. Conteñen obxectos, enlaces e mensaxes, e mostran a secuencia dinámica de mensaxes que flúen polos enlaces; adoitan capturar o comportamento dun só **caso de uso**. Hai dous tipos **isomorfos**: **diagramas de secuencia** (ordenación temporal de mensaxes) e **diagramas de colaboración** (organización estrutural de obxectos). Os **diagramas de secuencia** teñen unha **liña de vida** que representa a existencia dun obxecto e un **foco de control** que representa o tempo durante o cal un obxecto executa unha acción; os **diagramas de colaboración** teñen un **enlace** e un **número de secuencia**.

Unha **iteración** indica a repetición dunha mensaxe, e unha **bifurcación** representa un mensaxe cuxa execución depende da avaliación dunha condición.

Os diagramas de interacción poden modelar **colaboracións** e **escenarios** dentro dun caso de uso. Os **diagramas de secuencia** modelan escenarios ou visualizan iteracións e bifurcacións sinxelas; os de **colaboración** son preferibles para iteracións e bifurcacións complexas.

Diagramas de obxectos

Un **diagrama de obxectos** é unha captura dos obxectos nun sistema nun punto de tempo. Amosa **instancias** en lugar de clases. Pode empregarse para mostrar unha configuración exemplo de obxectos. Os seus elementos son **especificacións** de instancias e non instancias reais, xa que é

legal deixar baleiros atributos obrigatorios ou mostrar especificacións de instancias de clases abstractas.

Diagramas de estrutura composta

Os **diagramas de estrutura composta** permiten descompoñer unha clase en diferentes partes que representan a función que cumpre en tempo de execución unha instancia da clase. Pódense engadir **portos** á estrutura externa, que permiten agrupar as interfaces proporcionadas e requiridas en interaccións lóxicas que unha parte ten co mundo exterior.

Diagrama de actividades

Os **diagramas de actividades** combinan o diagrama de eventos, as técnicas de modelado de estados e as redes Petri. Resultan útiles en conexión co **fluxo de traballo** para describir o comportamento que ten unha gran cantidade de proceso paralelo. Permiten seleccionar a **orde** na que se farán as cousas.

Diagrama de compoñentes

Os **diagramas de compoñentes** representan como un sistema software está dividido en compoñentes e mostra as dependencias entre eles, é dicir, describe os elementos físicos do sistema e como se relacionan. Un compoñente debe ter un nome e pode conter adornos, etiquetas ou información adicional. Hai diferentes estereotipos: **executable, library, table, file** e **document.**

Diagramas de despregue

Os **diagramas de despregue** empréganse para modelar o hardware empregado nas implementacións de sistemas e as relacións entre os seus compoñentes. Mostran as relacións físicas entre os compoñentes hardware e software no sistema final. Describen a arquitectura física do sistema durante a execución e a súa topoloxía.

Diagramas de paquetes

Un **paquete** é unha parte do sistema. A subdivisión do sistema en paquetes debe basearse nunha **razón lóxica** (funcionalidade, obxectivo, implementación, etc.) Os paquetes poden conter a outros paquetes e hai un **paquete raíz** que contén ao sistema completo. Que exista unha **dependencia** entre dous paquetes quere dicir que existe unha dependencia entre algunha das unidades que conteñen, non necesariamente todas. Existe **xeralización** entre paquetes. Pódese determinar a súa **visibilidade** (privada por defecto).

Diagramas de estados

Os **diagramas de estados** empréganse para describir o comportamento dun sistema. Describen todos os estados posibles nos que pode entrar un obxecto particular e a maneira en que o seu estado cambia como resultado dos eventos que chegan a el. Na meirande parte das técnicas OO, o diagrama de estados amosa o comportamento **dun só obxecto** durante o seu ciclo de vida. Existen superestados que agrupan estados que comparten un acontecemento que os leva a un mesmo estado.

Diagramas de vista de interacción

Os **diagramas de vista de interacción** son **diagramas de comportamento**. Amosan unha certa **vista** sobre os aspectos dinámicos dos sistemas modelados. O **fluxo de control** modélase con elementos dos **diagramas de actividades**. O proceso comeza nun **nodo inicial** e remata nun **nodo terminal** para actividades. Poden representarse tamén **nodos de ramificación**. Poden usarse para sistemas non complexos.

Modelado estrutural avanzado

Hai diferentes niveis de abstracción: **conceptual** (representación dos conceptos do dominio), **especificación** (visión de interfaces do software) e **implementación** (exposición completa das clases implementadas).

En canto aos **atributos**, a **nivel conceptual** téñense atributos, a **nivel de especificación** pódense decidir e establecer os atributos, e a **nivel de implementación** tense un campo para gardar cada atributo.

As **operacións**, a **nivel conceptual** especifican as responsabilidades dunha clase, a **nivel de especificación** especifican a interface, e a **nivel de implementación** contémplanse operacións privadas e protexidas.

As **asociacións** a **nivel conceptual** denotan navegabilidade (posibilidade de ir dos obxectos dunha clase aos doutra), a **nivel de especificación** mostran **responsabilidades** e revelan **interfaces**, e a **nivel de implementación** permiten deducir unha **estrutura de datos**.

Os **mecanismos de extensibilidade** permiten ampliar UML de xeito controlado mediante **estereotipos** (creación de novos bloques), **valores etiquetados** (ampliación das propiedades dun bloque) e **restricións** (especificacións dunha nova semántica).

UML permite enunciar o significado dunha clase para calquera grao de formalismo: ao iniciar o desenvolvemento indícanse as **responsabilidades**, ao construír defínense **estrutura** e **comportamento** para asumir as responsabilidades, e finalmente modélanse **detalles**.

Os **clasificadores** son un mecanismo que describe características estruturais e de comportamento. Son elementos que poden ter **instancias**.

Hai catro **niveis de visibilidade**: **pública** (+), **protexida** (#), **privada** (-) e **paquete** (~). Por defecto é pública.

O **alcance** especifica se un atributo ou operación se manexa a nivel de cada instancia do clasificador ou se é común para todas. Os atributos e operacións con alcance de clasificador aparecen **subliñados**.

As **asociacións** son relacións de tipo **estrutural** que posúen navegabilidade, visibilidade dos roles e composición. Pode limitarse a visibilidade dun rol dende **obxectos externos** á asociación. As **clases asociación** son elementos con propiedades de asociación e de clase (atributos).

A **composición** é unha agregación con forte relación de pertenza entre o **todo** e as súas **partes**. As partes viven e morren co **todo**, que é quen xestiona a súa **creación** e **destrución**. Un obxecto só pode formar parte dun **todo**.

Unha **interface** é unha colección de operacións que deberá de implementar unha clase ou compoñente. **Non** especifican **estrutura** (non teñen atributos) nin **implementación** (non inclúen métodos). Poden participar en relacións de xeralización, asociación, dependencia e realización. A **realización** é unha relación semántica entre dous clasificadores na que un fai uso dun contrato

que o outro garante. Unha **interface ben estruturada** é **sinxela** e **completa, comprensible** e **manexable**.

Patróns de deseño

Os **patróns** son **solucións** de eficiencia demostrada a **problemas** de deseño **comúns**. Os sistemas que os usan son máis comprensibles e adaptables. Son útiles para identificar obxectos necesarios, determinar a súa granularidade, programar interfaces, reutilización (favorecer asociación fronte a herdanza) e deseño para o cambio.

Os **patróns de deseño** amosan **estrutura** e **comportamento** dunha sociedade de clases. Represéntanse como **colaboracións** con aspectos **estruturais** e de **comportamento**. Resolven problemas concretos de deseño e axudan a lograr solucións **flexibles** e **reutilizables**. Cada patrón nomea, explica e avalía un **deseño concorrente** en sistemas OO, axuda a lograr un bo deseño mediante **reutilización**, axuda a **elixir alternativas** que fan que o sistema sexa **reutilizable** e pode mellorar a **documentación** e o mantemento. Un patrón está composto de **nome**, **problema**, **solución** e **consecuencias**.

O **MVC** está composto por **modelo** (obxecto de aplicación), **vista** (representación do modelo) e **controlador** (resposta da interface á entrada do usuario). As vistas e os modelos están desacoplados, as vistas pódense aniñar e a resposta a unha vista pode cambiar sen alterar a súa representación visual.

Os **patróns de deseño** segundo o seu **propósito** poden ser de **creación** (ocúpanse do proceso de creación dos obxectos), **estruturais** (asociación de clases ou obxectos) ou de **comportamento** (interacción e reparto de responsabilidades entre clases e obxectos) e, segundo o seu **ámbito** poden ser de **clase** (relacións estáticas entre clases e subclases fixadas en tempo de compilación) ou de **obxecto** (relacións dinámicas que poden cambiar en tempo de execución).

Patróns de creación

Os **patróns de creación** abstraen o proceso de creación de obxectos. Poden ser de **clases** (usan herdanza para cambiar a clase da instancia a crear) ou de **obxectos** (delegan a creación de obxectos noutro obxecto). Encapsulan o coñecemento sobre clases concretas que emprega o sistema e ocultan como se crean e enlazan as instancias das clases e permiten configurar o sistema con obxectos *produto* que varían en estrutura e funcionalidade; o resto do sistema só coñece obxectos a través de interfaces.

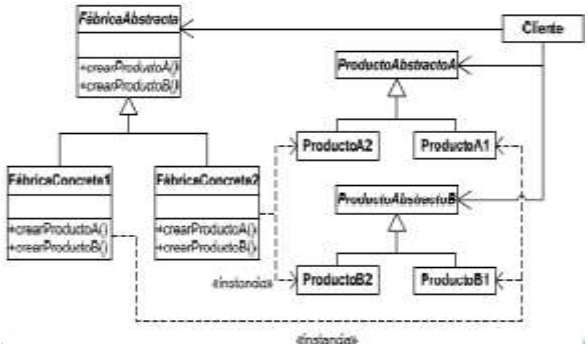
Abstract Factory

Proporciona unha interface para crear **familias** de obxectos dependentes ou relacionados sen especificar a súa clase concreta. **Usos:**

- Se un sistema debe de ser configurado para unha familia de produtos entre varias.
- Se cada familia está configurada para ser usada conxuntamente.
- Para proporcionar **librarías** de produtos dos que só se revelan as súas interfaces

Participantes:

- **FabricaAbstracta:** Declara unha interface para operacións que crean produtos abstractos.
- **FabricaConcreta:** Implementa operacións para crear produtos concretos.
- **ProdutoAbstracto:** Declara unha interface do tipo produto.
- **ProdutoConcreto:** Define un produto a ser creado pola fábrica correspondente.
- **Cliente:** Usa interfaces de clases abstractas.



Vantaxes e inconvenientes:

- Illamento entre clientes e clases de implementación.
- Facilidade de cambio de familias.
- Consistencia.
- Dificultade de introducir novos produtos.

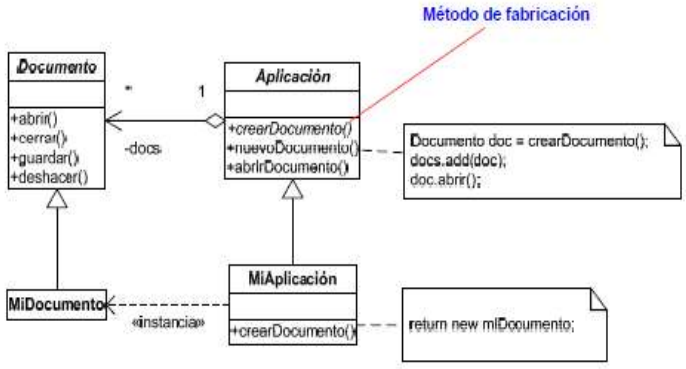
Factory Method

Define unha interface para crear obxectos deixando a subclases que decidan que clase instanciar. Permite que unha clase delegue en subclases a creación de obxectos. **Úsase cando**:

- Unha clase non pode prever a clase de obxectos que debe crear.
- Unha clase quere delegar nas súas subclases a creación de obxectos.

Participantes:

- **Produto:** Define interface dos obxectos creados.
- **ProdutoConcreto:** Implementa a interface Produto.
- **Creador:** Declara o método de fabricación, que devolve un obxecto Produto.
- **CreadorConcreto:** Redefine o método de fabricación para devolver unha instancia de ProdutoConcreto.



Vantaxes e inconvenientes:

- Código de usuario só trata coa interface Produto.
- Clientes poden ter que herdar de Creador para crear un determinado ProdutoConcreto.
- Dota a subclases de enganche para proveer versión estendida dun obxecto.
- Pode conectar xerarquías paralelas.

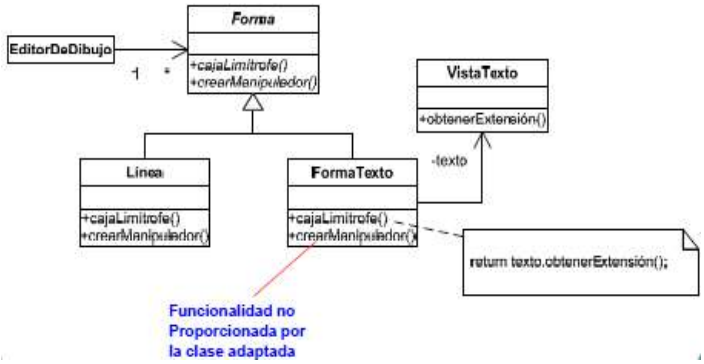
Patróns estruturais

Combinan clases e obxectos para lograr estruturas maiores. Os patróns de **clases** usan herdanza para compoñer interfaces ou implementacións e os de **obxectos** describen formas de ensamblar obxectos para obter funcionalidade.

Adapter

Convirte a interface dunha clase noutra que agardan os clientes. Permite cooperar a clases con interfaces incompatibles. **Úsase cando:**

- Se quere usar unha clase existente e a súa interface non concorda coa necesaria.
- Se quere crear unha clase reutilizable que coopere con clases non previstas.
- É necesario usar varias subclases existentes e a herdanza non é práctica para adaptar a súa interface.



Participantes:

- **Obxectivo:** Define a interface específica do dominio que emprega o cliente.
- **Cliente:** Colabora con obxectos que se axustan á interface Obxectivo.
- **Adaptado:** Define interface existente que precisa ser adaptada.
- **Adaptador:** Adapta a interface Adaptado á interface Obxectivo.

Vantaxes e inconvenientes:

- Permite ao mesmo Adaptador funcionar con varios Adaptados.
- Máis difícil redefinir comportamento de Adaptado.

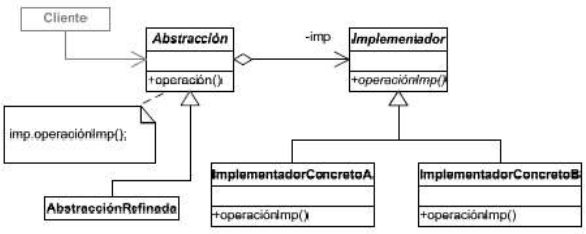
Bridge

Separa unha abstracción da súa implementación para que ambas poidan variar independentemente. Está pensado para cambiar a interface dun obxecto existente. **Motivación:**

- Necesidade de varias implementacións da mesma abstracción.
- O enfoque mediante herdanza é pouco flexible.

Usos:

- Evitar vínculo permanente entre abstracción e implementación.
- Ampliar abstracción e implementación mediante herdanza.
- Impedir que cambios na implementación teñan impacto nos clientes.
- Compartir implementación entre varios clientes ocultando detalles.



Participantes:

- **Abstracción:** Define interface da abstracción e referencia a un obxecto de tipo Implementador.
- **AbstracciónRefinada:** Estende interface definida por Abstracción
- **Implementador:** Define interface para clases de implementación.
- **ImplementadorConcreto:** Implementa a interface Implementador.

Vantaxes e inconvenientes:

- Desacopla interface e implementación.
- Permite estender independentemente Abstracción e Implementador.
- Oculta detalles de implementación aos clientes.

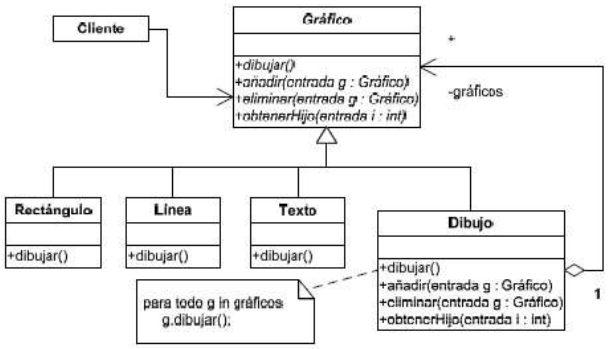
Diferenza entre Adapter e Bridge

- **Adapter** resolve incompatibilidades entre interfaces existentes sen preocuparse de como poderían evolucionar independentemente. Úsase despois do deseño.
- **Bridge** une abstracción e implementación. Úsase antes do deseño.

Composite

Compón obxectos en estruturas de árbore para representar **xerarquías** e permite tratar de xeito uniforme a obxectos individuais e compostos. **Úsase para:**

- Representar xerarquías parte-todo.
- Obviar diferenzas entre obxectos e composicións deles.



Participantes:

- **Compoñente:** Declara interface de obxectos da agregación e implementa comportamento predeterminado.
- **Folla:** Establece comportamento de obxectos primitivos.
- **Composto:** Define comportamento de compoñentes con fillos e implementa as operacións de acceso a fillos.
- **Cliente:** Manipula obxectos a través da interface Compoñente.

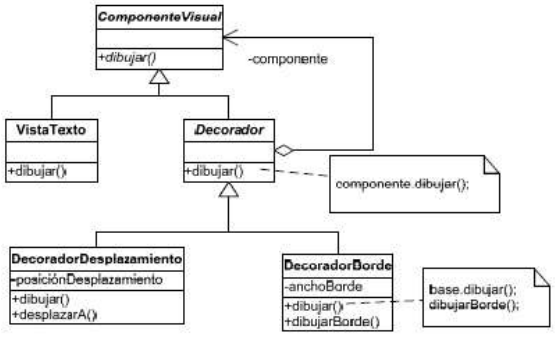
Vantaxes e inconvenientes:

- Permite agregación recursiva.
- Uniformiza acceso a compoñentes.
- Facilita a adición de novos tipos de compoñentes.
- Difícil restrinxir compoñentes permitidos para un composto.

Decorator

Asigna ou retira responsabilidades a un obxecto dinamicamente. Proporciona unha alternativa á herdanza para ampliar a súa funcionalidade. Adoita usarse con Composite. **Usos:**

- Cando non é viable a extensión mediante herdanza.



Participantes:

- **Compoñente:** Define interface dos obxectos aos que se poden engadir responsabilidades.
- **CompoñenteConcreto:** Define o obxecto ao que se lle poden engadir responsabilidades.
- **Decorador:** Referencia a Compoñente e axústase á súa interface.
- **DecoradorConcreto:** Engade responsabilidades ao Compoñente.

Vantaxes e inconvenientes:

- Maior flexibilidade que a herdanza.
- Evita clases cargadas de funcións na parte superior da xerarquía.
- Un decorador e o seu compoñente non son idénticos.
- Pode dar lugar a obxectos pequenos moi semellantes.

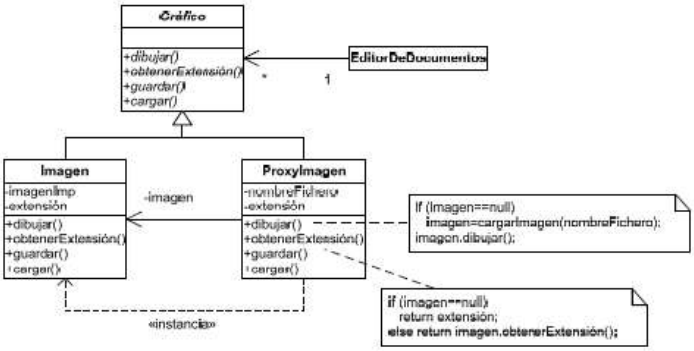
Diferenzas entre Composite e Decorator

- **Decorator** engade responsabilidades sen crear subclases.
- **Composite** uniformiza acceso a obxectos individuais e composicións deles.

Proxy

Subministra un representante ou substituto dun obxecto para controlar o acceso a el. A súa interface é idéntica á do obxecto representado. **Aplicacións:**

- **Proxy remoto** Representante local dun obxecto noutro espazo de direccións.
- **Proxy virtual:** Crea obxectos de alto custe por encargo.
- **Proxy de protección:** Controla acceso ao obxecto.
- **Referencia intelixente:** Substituto dunha referencia, fai operacións adicionais. Conta o número de referencias ao obxecto real para que se poida liberar cando ninguén o apunte. Carga un obxecto persistente en memoria ao ser referenciado por primeira vez e xestiona bloqueos do obxecto real para impedir intentos simultáneos de modificación.



Participantes:

- **Proxy:** Controla o acceso ao obxecto e pode ser responsable de crealo e borralo.
- **Suxeito:** Define a interface común para SuxeitoReal e Proxy.
- **SuxeitoReal:** Obxecto real representado.

Vantaxes e inconvenientes:

- A indirección adicional introducida ten varios usos posibles.
- Creación diferida de copias

Diferenzas entre Decorator e Proxy

- **Decorator** asigna propiedades dinámica e recursivamente.
- **Proxy** proporciona un substituto dun obxecto para non acceder directamente a el.

Patróns de comportamento

Teñen que ver con algoritmos e asignación de responsabilidades. Describen fluxos de control complexos. Os patróns de **clases** usan herdanza para distribuír comportamento, e os de **obxectos** describen cooperación para realizar tarefas que os obxectos non poden realizar en solitario.

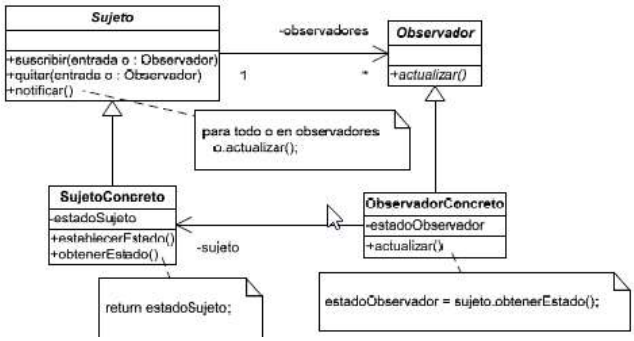
Observer

Define e mantén unha dependencia de un a moitos entre obxectos. Notifica o cambio de estado dun obxecto aos observadores que dependen del e sincronizan o seu estado co do obxecto. **Usos**

- Un aspecto dunha abstracción dependa doutro.
- Un cambio nun obxecto requira cambiar outros e non se saiba cantos.
- Un obxecto deba ser capaz de notificar a outros sen facer suposicións sobre quen son.

Participantes:

- **Suxeito:** Coñece aos seus observadores e proporciona interface para engadilos e quitalos.
- **Observador:** Define interface para actualizar obxectos concretos ante cambios nun suxeito.
- **SuxeitoConcreto:** Almacena estados de interese para os seus observadores concretos e notifícalles cambios nese estado.
- **ObservadorConcreto:** Referencia a un suxeito concreto co que sincroniza o seu estado.



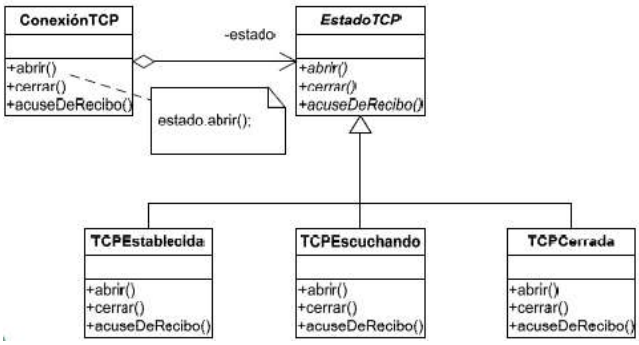
Vantaxes e inconvenientes:

- Permite modificar suxeitos e observadores de xeito independente.
- O suxeito non precisa saber a clase concreta de ningún observador.
- Capacidade de comunicación por difusión.
- Observadores non se coñecen entre si.

State

Permite que un obxecto modifique o seu comportamento cada vez que cambia o seu estado. Parecerá que cambia a clase dese obxecto. **Úsase cando:**

- O comportamento dun obxecto depende do seu estado e cambia en tempo de execución.
- Os seus métodos presentan estruturas condicionais con ramas de bloques cuxa execución depende do estado do obxecto.



Participantes:

- **Contexto:** Define interface de interese para os clientes.
- **Estado:** Declara interface para encapsular comportamento asociado a estados do Contexto.
- **EstadoConcreto:** Implementa comportamento asociado a un estado do Contexto.

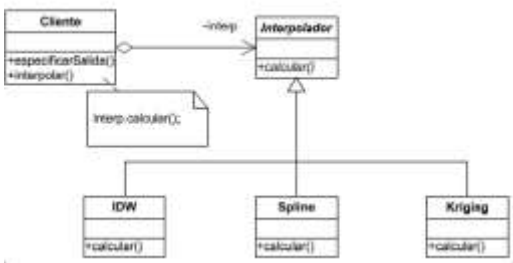
Vantaxes e inconvenientes:

- Sitúa nun só obxecto o comportamento asociado a un estado.
- Protección fronte a estados inconsistentes.
- Estados poden compartirse entre diferentes contextos se non teñen atributos.

Strategy

Define unha familia de algoritmos e fainos intercambiables. Permite configurar unha clase cun comportamento de entre varios posibles e que un algoritmo varíe con independencia dos clientes que o usan. **Úsase cando:**

- Existen moitas clases relacionadas que difiren só no seu comportamento.
- Se precisan variantes dun algoritmo.
- Un algoritmo usa datos que os clientes non deberían coñecer.
- Unha clase define moitos comportamentos que se representan como múltiples sentenzas condicionais.



Participantes:

- **Estratexia:** Declara interface común a todo algoritmo.
- **EstratexiaConcreta:** Implementa un algoritmo particular.
- **Contexto:** Referencia a unha estratexia e configúrase cunha estratexia concreta.

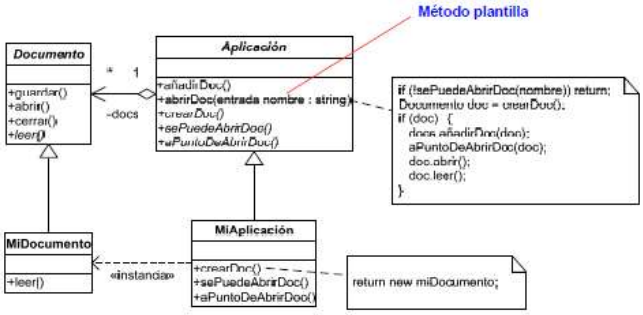
Vantaxes e inconvenientes:

- Define familia de algoritmos intercambiables con diferentes solucións de compromiso.
- Permite modificar un algoritmo con independencia do seu contexto.
- Elimina sentencias condicionais.
- Clientes deben coñecer estratexias dispoñibles.
- Estratexias máis simples poden non usar toda a información recibida a través da súa interface.
- Aumenta o número de obxectos.

Template Method

Define o esqueleto dun algoritmo delegando algúns pasos en subclases e permite redefinir algúns pasos do mesmo sen cambiar a súa estrutura. Cada paso invoca a unha operación abstracta ou concreta. As subclases completan o algoritmo implementando operacións abstractas. **Aplicación:**

- Para implementar as partes dun algoritmo que non cambian e deixar que as subclases definan o comportamento variable.
- Cando o comportamento repetido deba ser refactorizado nunha clase común.
- Para controlar extensións de subclases mediante operacións de enganche.



Participantes:

- **ClaseAbstracta:** Declara operacións primitivas abstractas definidas polas subclases para implementar os pasos dun algoritmo. Implementa un método modelo que define o esqueleto do algoritmo.
- **ClaseConcreta:** Implementa operacións primitivas para realizar os pasos.