|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **GRASP - (General Responsability Assignment Software Patterns) décrivent des principes fondamentaux d’assignation des responsabilités à des objets. Ils sont utilisés comme guide pour assigner correctement les responsabilités aux différentes classes lors du passage vers la conception** | | | | |
| **Patrons** | **Intention** | **Applicabilité** | **Conséquences** | **Exemple** |
| Expert | Donner la responsabilité à l’objet le mieux en mesure d’y satisfaire, | La prise en charge d’une responsabilité requiert de l’information distribuée parmi plusieurs objets ou classes, ce qui donne lieu à plusieurs « experts partiels » qui collaborent pour remplir complètement la responsabilité. | +Favorise l’encapsulation, puisque les objets utilisent leur propre information pour satisfaire la responsabilité,  +Supporte un couplage faible entre les classes, ce qui mène à des systèmes plus robustes et plus faciles à maintenir,  + Distribue le comportement parmi les classes qui possèdent l’information nécessaire: ce qui encourage des définitions de classes plus cohérentes et plus faciles à comprendre. |  |
| Créateur | Donner la responsabilité de créer un objet à un autre objet directement impliqué dans une relation avec l’objet créé | Le patron Créateur guide dans l’assignation de la responsabilité de création d’objets, une tâche très commune dans les systèmes orientés-objets. L’idée de base du patron Créateur consiste à trouver une classe qui a déjà d’excellentes raisons d’être associée à la classe à créer. | + Supporte un couplage faible entre les classes, ce qui mène à moins de dépendances et plus de possibilités de réutilisation. Le couplage n’est probablement pas augmenté, puisque le Créateur a déjà d’excellentes raisons d’être connecté à la classe des objets qu’il crée. | Transaction: le patron Créateur suggère de chercher une classe qui contient, enregistre, etc. des transactions. Puisque Compte enregistre des transactions, Compte pourrait être un bon candidat: |
| Couplage Faible | Assigner les responsabilités de façon à ce que le couplage reste faible | Le couplage est une mesure du niveau de connexion d’une classe avec les autres classes du système. Une classe ayant un Faible couplage n’est pas connectée à un « trop » grand nombre d’autres classes. Le Faible couplage est un principe de conception qui doit toujours être présent à l’esprit des concepteurs, il s’agit d’un patron d’évaluation d’une conception. | Le Faible couplage mène à la conception de classes qui sont plus indépendantes, ce qui réduit l’impact de changements, et qui sont plus réutilisables, ce qui améliore les possibilités de gains de productivité.  – Le couplage ne peut pas être considéré de façon isolée, mais doit être évalué en même temps que d’autres principes comme Expert et Forte Cohésion dans l’évaluation d’une conception,  – Le couplage n’est peut-être pas important si la réutilisation n’est pas un objectif.  – Le cas extrême du Faible couplage, où une classe n’est connectée à pratiquement aucune classe n’est pas désirable, puisqu’il mène à des classes ayant une très faible cohésion, qui assume toutes les responsabilités à l’interne. |  |
| Couplage Élevée | Assigner les responsabilités de façon à ce que la cohésion reste élevée | La cohésion est une mesure du niveau de lien qui existe entre les différentes fonctions d’une classe. Si toutes les responsabilités d’une classe sont reliées les unes aux autres, la cohésion de la classe est forte. La Cohésion élevée est un principe de conception qui doit toujours être présent à l’esprit des concepteurs, il s’agit d’un patron d’évaluation d’une conception | + La cohésion élevée d’une classe augmente la clarté et la facilité de compréhension de la classe,  + La cohésion élevée facilite la maintenance et les améliorations,  + Aide à maintenir le couplage bas,  + La granularité fine de l’assignation des responsabilités augmente la possibilité de réutilisation, étant donné qu’une classe ayant un haut niveau de cohésion peut être utilisé dans un but bien précis. |  |
| Contrôleur | Assigner la responsabilité de gérer les messages d’évènements du système à une classe spécialisée du modèle, | Le patron Contrôleur s’applique dans pratiquement tous les systèmes orientés objet qui doivent traiter des événements externes impliquant par exemple un usager ou des systèmes logiciels ou physiques externes (senseurs, commutateurs, etc.). Dans tous les cas où une conception orientée objet est utilisée, une classe Contrôleur doit être choisie pour traiter les événements. Le patron Contrôleur constitue un guide pour choisir une classe acceptable. | + Centralise la gestion des événements dans une classe bien identifiée,  + Augmente les possibilités de réutilisation en s’assurant que les processus liés au domaine d’application sont traités au niveau de la couche du modèle et non de la couche de présentation,  + Permet de s’assurer de la cohérence séquentielle des événements en fournissant une classe unique pour traiter les événements, qui peut s’assurer qu’ils se produisent dans le bon ordre.  – Il est facile d’assigner trop de responsabilités au Contrôleur, ce qui mène à une classe ayant peu de cohésion, et difficile à comprendre,  – Lorsqu’un Contrôleur assume trop de responsabilités, on peut le décharger en créant des Contrôleurs spécialisés, ou en s’assurant que le Contrôleur délègue la gestion de chaque événement à un objet spécialisé. | Deux approches sont envisageables pour solutionner le problème d’un Contrôleur trop complexe:  1. Déléguer le traitement de certains événements à d’autres contrôleurs: patron « Chain of responsability »,  2. Traiter chaque événement système comme un objet autonome: patron « Command » |
| Polymorphisme | Utiliser des interfaces pour traiter des alternatives qui dépendent du type des objets, | Un autre principe fondamental de l’approche orientée-objet.  Dès que le comportement d’objets reliés les uns aux autres doit varier en fonction du type concret de l’objet.  Lorsqu’il serait nécessaire de tester le type concret de l’objet et d’utiliser des énoncés conditionnels ou des énoncés switch/case | + Permet d’ajouter facilement les extensions requises pour de nouvelles variations.  + De nouvelles implémentations peuvent être ajoutées sans affecter les clients.  – Permet d’ajouter des interfaces ou des points de variation en prévision de développements futurs qui ne se réaliseront peut-être pas.  – Ajouter seulement les points de variation pour des alternatives qui sont confirmées par la conception du système et éviter d’ajouter des points de variation superflus. |  |
| Indirection | Assigner des responsabilités à un objet intermédiaire pour éviter que des classes soient couplées, | Vieil adage:  « La plupart des problèmes en informatique peuvent être solutionnés par un niveau supplémentaire d’indirection. »  Et son corolaire:  « La plupart des problèmes de performance peuvent être solutionnés en éliminant une couche d’indirection. » | + Permet de réduire le couplage entre des classes ou des composants,  – Augmente le nombre de classes,  – Réduit la performance. |  |
| Pure Fabrication | Assigner un ensemble de responsabilité à une classe créée artificiellement, | Lorsque l’on veut regrouper dans une classe un ensemble de méthodes qui sont liées entre-elles, mais qui ne correspondent pas directement à un concept du domaine d’application. Lorsqu’un ensemble de comportements doivent être extrait d’une classe et regroupé afin de maintenir une cohésion élevée. | + Permet de factoriser dans une nouvelle classe un ensemble de comportements qui méritent d’être regroupés,  + Permet de maintenir une cohésion élevée de la classe d’où sont extraits les comportements et de la nouvelle classe créée,  – Créer trop de classes strictement basées sur le comportement tend vers une décomposition plus fonctionnelle qu’orientée-objet,  – Une surabondance de classes fabriquées tend à éloigner les données et les comportements qui leur sont associés, ce qui contredit le patron Expert et affecte négativement le couplage. |  |
| Variation protégée | Identifier des points d’instabilité ou de variation et développer des interfaces stables autour de ces points. | Il s’agit d’un principe de conception fondamental, très important dans n’importe quel contexte de résolution de problème. Principe de base derrière des principes tels que l’encapsulation des données, les interfaces, le polymorphisme et les indirections. | Simplifie l’ajout d’extensions nécessaires pour de nouvelles variations,  + De nouvelles implémentations peuvent être ajoutées sans affecter les clients,  + Réduit le couplage,  + Permet de réduire l’impact ou le coût de changements,  – Favorise l’ajout de points d’évolution pour des alternatives qui ne se concrétiseront peut-être pas,  – Favorise l’ajout de flexibilité non requise. |  |