Intergiciel et Construction d'Applications Réparties

Chapitre 2

Patrons et canevas pour l'intergiciel

Ce chapitre présente les grands principes de conception des systèmes intergiciels, ainsi que quelques patrons élémentaires récurrents dans toutes les architectures intergicielles. Divers patrons plus élaborés peuvent être construits en étendant et en combinant ces constructions de base. Le chapitre débute par une présentation des principes architecturaux et des éléments constitutifs des systèmes intergiciels, notamment les objets répartis et les organisations multi-couches. Il continue par une discussion des patrons de base relatifs aux objets répartis. Le chapitre se termine par une présentation des patrons liés à la séparation des préoccupations, qui comprend une discussion des techniques de réalisation pour l'intergiciel réflexif.

2.1 Services et interfaces

Un système matériel et/ou logiciel est organisé comme un ensemble de parties, ou composants[1]. Le système entier, et chacun de ses composants, remplit une fonction qui peut être décrite comme la fourniture d'un *service*. Selon une définition tirée de [Bieber and Carpenter 2002], « un service est un comportement défini par contrat, qui peut être réalisé et fourni par tout composant pour être utilisé par tout composant, sur la base unique du contrat ».

Pour fournir ses services, un composant repose généralement sur des services qu'il demande à d'autres composants. Par souci d'uniformité, le système entier peut être considéré comme un composant, qui interagit avec un environnement externe spécifié ; le service fourni par le système repose sur des hypothèses sur les services que ce dernier reçoit de

[1]Dans ce chapitre, nous utilisons le terme de *composant* dans un sens non-technique, pour désigner une unité de décomposition d'un système.

son environnement[2].

La fourniture de services peut être considérée à différents niveaux d'abstraction. Un service fourni est généralement matérialisé par un ensemble d'interfaces, dont chacune représente un aspect du service. L'utilisation de ces interfaces repose sur des patrons élémentaires d'interaction entre les composants du système. Dans la section 2.1.1, nous passons brièvement en revue ces patrons d'interaction. Les interfaces sont discutées dans la section 2.1.2, et les contrats sont l'objet de la section 2.1.3.

2.1.1 Mécanismes d'interaction de base

Les composants interagissent via un système de communication sous-jacent. Nous supposons acquises les notions de base sur la communication et nous examinons quelques patrons d'interaction utilisés pour la fourniture de services.

La forme la plus simple de communication est un événement transitoire asynchrone (Figure 2.1a). Un composant A (plus precisément, un *thread* s'exécutant dans le composant A) produit un événement (c'est-à-dire envoie un message élémentaire à un ensemble spécifié de destinataires), et poursuit son exécution. Le message peut être un simple signal, ou peut porter une valeur. L'attribut « transitoire » signifie que le message est perdu s'il n'est pas attendu. La réception de l'événement par le composant B déclenche une réaction, c'est-à-dire lance l'exécution d'un programme (le traitant) associé à cet événement. Ce mécanisme peut être utilisé par A pour demander un service à B, lorsqu'aucun résultat n'est attendu en retour ; ou il peut être utilisé par B pour observer ou surveiller l'activité de A.

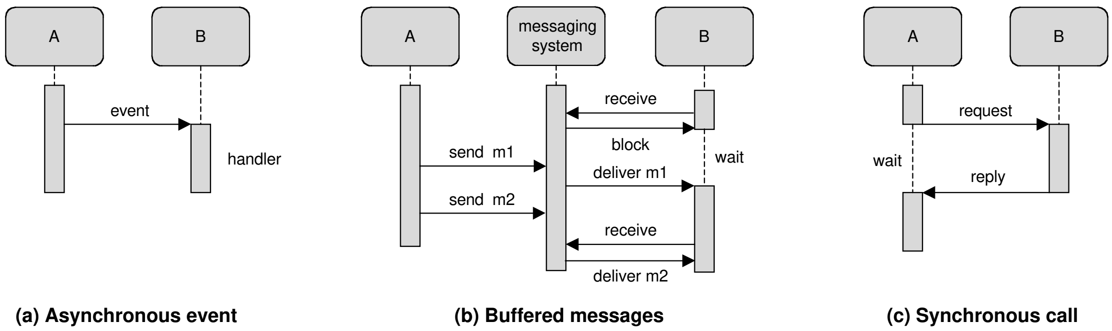

Figure 2.1 – Quelques mécanismes de base pour l'interaction

Une forme de communication plus élaborée est le passage asynchrone de messages persistants (2.1b). Un message est un bloc d'information qui est transmis d'un émetteur à un récepteur. L'attribut « persistant » signifie que le système de communication assure un rôle de tampon : si le récepteur attend le message, le système de communication le lui délivre ; sinon, le message reste disponible pour une lecture ultérieure.

Un autre mécanisme courant est l'appel synchrone (2.1c), dans lequel A (le client d'un service fourni par B) envoie un message de requête à B et attend une réponse. Ce patron

[2]Par exemple un ordinateur fournit un service spécifié, à condition de disposer d'une alimentation électrique spécifiée, et dans une plage spécifiée de conditions d'environnement, telles que température, humidité, etc.

est utilisé dans le RPC (voir chapitre 1, section 1.3).

Les interactions synchrone et asynchrone peuvent être combinées, par exemple dans diverses formes de « RPC asynchrone ». Le but est de permettre au demandeur d'un service de continuer son exécution après l'envoi de sa requête. Le problème est alors pour le demandeur de récupérer les résultats, ce qui peut être fait de plusieurs manières. Par exemple, le fournisseur peut informer le demandeur, par un événement asynchrone, que les résultats sont disponibles ; ou le demandeur peut appeler le fournisseur à un moment ultérieur pour connaître l'état de l'exécution.

Il peut arriver que la fourniture d'un service par B à A repose sur l'utilisation par B d'un service fourni par A (le contrat entre fournisseur et client du service engage les deux parties). Par exemple, dans la figure 2.2a, l'exécution de l'appel depuis A vers B repose sur un *rappel* (en anglais *callback*) depuis B à une fonction fournie par A. Sur cet exemple, le rappel est exécuté par un nouveau *thread*, tandis que le *thread* initial continue d'attendre la terminaison de son appel.

Les exceptions sont un mécanisme qui traite les conditions considérées comme sortant du cadre de l'exécution normale d'un service : pannes, valeurs de paramètres hors limites, etc. Lorsqu'une telle condition est détectée, l'exécution du service est proprement terminée (par exemple les ressources sont libérées) et le contrôle est rendu à l'appelant, avec une information su la nature de l'exception. Une exception peut ainsi être considérée comme un « rappel à sens unique ». Le demandeur du service doit fournir un traitant pour chaque exception possible.

La notion de rappel peut encore être étendue. Le service fourni par B à A peut être demandé depuis une source extérieure, A fournissant toujours à B une ou plusieurs interfaces de rappel. Ce patron d'interaction (Figure 2.2b) est appelé *inversion du contrôle*, parce que le flot de contrôle va de B (le fournisseur) vers A (le demandeur). Ce cas se produit notamment lorsque B « contrôle » A, c'est-à-dire lui fournit des services d'administration tels que la surveillance ou la sauvegarde persistante ; dans cette situation, la demande de service a une origine externe (elle est par exemple déclenchée par un événement extérieur tel qu'un signal d'horloge).

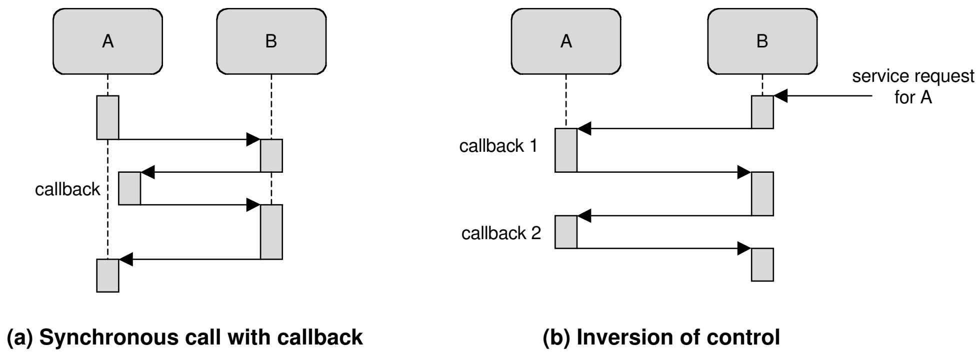

Figure 2.2 – Inversion du contrôle

Les interactions ci-dessus sont discrètes et n'impliquent pas explicitement une notion de temps autre que l'ordre des événements. Les échanges continus nécessitent une forme de synchronisation en temps réel. Par exemple les données multimédia sont échangées via des *flots de données*, qui permettent la transmission continue d'une séquence de données soumise à des contraintes temporelles.

2.1.2 Interfaces

Un service élémentaire fourni par un composant logiciel est défini par une *interface*, qui est une description concrète de l'interaction entre le demandeur et le fournisseur du service. Un service complexe peut être défini par plusieurs interfaces, dont chacune représente un aspect particulier du service. Il y a en fait deux vues complémentaires d'une interface.

- la vue d'usage : une interface définit les opérations et structures de données utilisées pour la fourniture d'un service ;
- la vue contractuelle : une interface définit un contrat entre le demandeur et le fournisseur d'un service.

La définition effective d'une interface requiert donc une représentation concrète des deux vues, par exemple un langage de programmation pour la vue d'usage et un langage de spécification pour la vue contractuelle.

Rappelons qu'aussi bien la vue d'usage que la vue contractuelle comportent deux partenaires[3]. En conséquence, la fourniture d'un service implique en réalité *deux* interfaces : l'interface présentée par le composant qui fournit un service, et l'interface attendue par le client du service. L'interface fournie (ou serveur) doit être « conforme » à l'interface requise (ou client), c'est-à-dire compatible avec elle ; nous revenons plus loin sur la définition de la conformité.

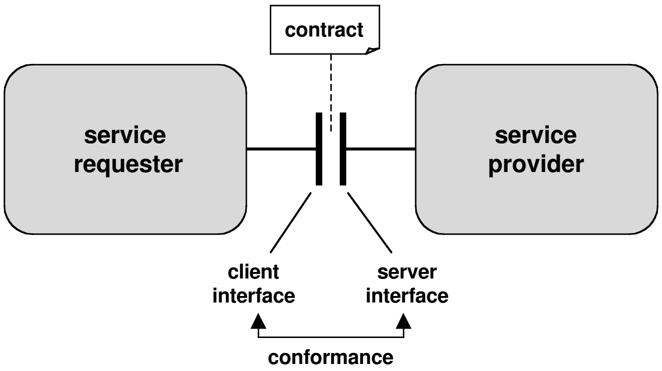

Figure 2.3 – Interfaces

La représentation concrète d'une interface, fournie ou requise, consiste en un ensemble d'opérations, qui peut prendre des formes diverses, correspondant aux patrons d'interaction décrits en 2.1.1.

[3]Certaines formes de service mettent en jeu plus de deux partenaires, par exemple un fournisseur avec demandeurs multiples, etc. Il est toujours possible de décrire de telles situations par des relations bilatérales entre un demandeur et un fournisseur, par exemple en définissant des interfaces virtuelles qui multiplexent des interfaces réelles, etc.

– procédure synchrone ou appel de méthode, avec paramètres et valeur de retour ; accès à un attribut, c'est-à-dire à une structure de données (cette forme peut être convertie dans la précédente au moyen de fonctions d'accès (en lecture, en anglais *getter* ou en écriture, en anglais *setter*) sur les éléments de cette structure de données) ;
– appel de procédure asynchrone ;
– source ou puits d'événements ;
– flot de données fournisseur (*output channel*) ou récepteur (*input channel*) ;

Le contrat associé à l'interface peut par exemple spécifier des contraintes sur l'ordre d'exécution des opérations de l'interface (par exemple ouvrir un fichier avant de le lire). Les diverses formes de contrats sont examinées dans la section 2.1.3.

Diverses notations, appelées Langages de Description d'Interface (IDL), ont été conçues pour décrire formellement des interfaces. Il n'y a pas actuellement de modèle unique commun pour un IDL, mais la syntaxe de la plupart des IDLs existants est inspirée par celle d'un langage de programmation procédural. Certains langages (par exemple Java, C#) comportent une notion d'interface et définissent donc leur propre IDL. Une définition d'interface typique spécifie la signature de chaque opération, c'est-à-dire son nom, son type et le mode de transmission de ses paramètres et valeurs de retour, ainsi que les exceptions qu'elle peut provoquer à l'exécution (le demandeur doit fournir des traitants pour ces exceptions).

La représentation d'une interface, avec le contrat associé, définit complètement l'interaction entre le demandeur et le fournisseur du service représenté par l'interface. En conséquence, ni le demandeur ni le fournisseur ne doit faire d'autre hypothèse sur son partenaire que celles explicitement spécifiées dans l'interface. En d'autre termes, tout ce qui est au-delà de l'interface est vu par chaque partenaire comme une « boîte noire ». C'est le *principe d'encapsulation*, qui est un cas particulier de la séparation des préoccupations. Le principe d'encapsulation assure l'indépendance entre interface et réalisation, et permet de modifier un système selon le principe « je branche et çà marche »(*plug and play*) : un composant peut être remplacé par un autre à condition que les interfaces entre le composant remplacé et le reste du système restent compatibles.

2.1.3 Contrats et conformité

Le contrat entre le fournisseur et le client d'un service peut prendre diverses formes, selon les propriétés spécifiées et selon l'expression plus ou moins formelle de la spécification. Par exemple, le terme *Service Level Agreement* (SLA) est utilisé pour un contrat légal entre le fournisseur et le client d'un service global de haut niveau (par exemple entre un fournisseur d'accès à l'Internet (en anglais *Internet Service Provider*, ou ISP) et ses clients.

D'un point de vue technique, différentes sortes de propriétés peuvent être spécifiées. D'après [Beugnard et al. 1999], on peut distinguer quatre niveaux de contrats.

– Le niveau 1 s'applique à la forme des opérations, généralement en définissant des *types* pour les opérations et les paramètres. Cette partie du contrat peut être statiquement verifiée.
– Le niveau 2 s'applique au comportement dynamique des opérations de l'interface, en spécifiant la sémantique de chaque opération.
– Le niveau 3 s'applique aux interactions dynamiques entre les opérations d'une interface, en spécifiant des contraintes de synchronisation entre les exécutions de ces

opérations. Si le service est composé de plusieurs interfaces, il peut aussi exister des contraintes entre l'exécution d'opérations appartenant à différentes interfaces.
– Le niveau 4 s'applique aux propriétés extra-fonctionnelles, c'est-à-dire à celles qui n'apparaissent pas explicitement dans l'interface. Le terme de « Qualité de Service » (QoS) est aussi utilisé pour ces propriétés, qui comprennent performances, securité, disponibilité, etc.

Notons encore que le contrat s'applique dans les deux sens, à tous les niveaux : il engage donc le demandeur aussi bien que le fournisseur. Par exemple, les paramètres passés lors d'un appel de fonction sont contraints par leur type ; si l'interface comporte un rappel, la procédure qui réalise l'action correspondante côté client doit être fournie (cela revient à spécifier une procédure comme paramètre).

L'essence d'un contrat d'interface est exprimée par la notion de conformité. Une interface *I2* est dite *conforme* à une interface *I1* si un composant qui réalise toute méthode spécifiée dans *I2* peut partout être utilisé à la place d'un composant qui réalise toute méthode spécifiée dans *I1*. En d'autres termes, *I2* est conforme à *I1* si *I2* satisfait le contrat de *I1*.

La conformité peut être vérifiée à chacun des quatre niveaux définis ci-dessus. Nous les examinons successivement.

Contrats syntaxiques

Un contrat syntaxique est fondé sur la forme des opérations. Un tel contrat s'exprime couramment en termes de types. Un *type* définit un prédicat qui s'applique aux objets[4] de ce type. Le type d'un objet X est noté $T(X)$. La notion de conformité est exprimée par le *sous-typage* : si *T2* est un sous-type de *T1* (noté $T2 \sqsubseteq T1$), tout objet de type *T2* est aussi un objet de type *T1* (en d'autre termes, un objet de type *T2* peut être utilisé partout où un objet de type *T1* est attendu). La relation de sous-typage ainsi définie est appelée sous-typage *vrai*, ou conforme.

Considérons des interfaces définies comme un ensemble de procédures. Pour de telles interfaces, le sous-typage conforme est défini comme suit : une interface *I2* est un sous-type d'une interface de type *I1* (noté $T(I2) \sqsubseteq T(I1)$) si *I2* a au moins le même nombre de procédures que *I1* (elle peut en avoir plus), et si pour chaque procédure définie dans *I1* il existe une procédure conforme dans *I2*. Une procédure *Proc2* est dite conforme à une procédure *Proc1* lorsque les relations suivantes sont vérifiées entre les signatures de ces procédures.

– *Proc1* et *Proc2* ont le même nombre de paramètres et valeurs de retour (les exceptions déclarées sont considérées comme des valeurs de retour).
– pour chaque valeur de retour *R1* de *Proc1*, il existe une valeur de retour correspondante *R2* de *Proc2* telle que $T(R2) \sqsubseteq T(R1)$ (relation dite *covariante*).
– pour chaque paramètre d'appel *X1* de *Proc1*, il existe un paramètre d'appel correspondant *X2* de *Proc2* tel que $T(X1) \sqsubseteq T(X2)$ (relation dite *contravariante*).

Ces règles illustrent un principe général de possibilité de substitution : une entité *E2* peut être substituée à une autre entité *E1* si *E2* « fournit au moins autant et requiert au

[4]Ici le terme d'*objet* désigne toute entité identifiable dans le présent contexte, par exemple une variable, une procédure, une interface, un composant.

plus autant » que *E1*. Ici les termes « fournit » et « requiert » doivent être être adaptés à chaque situation spécifique (par exemple dans un appel de procédure , les paramètres d'appel sont « requis » et le résultat est « fourni »). La relation d'ordre qu'impliquent les termes « au plus autant » et « au moins autant » est la relation de sous-typage.

Notons que la relation de sous-typage définie dans la plupart des langages de programmation ne satisfait généralement pas la contravariance des types de paramètres et n'est donc pas un sous-typage vrai. Dans un tel cas (qui est par exemple celui de Java), des erreurs de conformité peuvent échapper à la détection statique et doivent être capturées par un test à l'exécution.

La notion de conformité peut être étendue aux autre formes de définitions d'interface, par exemple celles contenant des sources ou puits d'événements, ou des flots de données (*streams*).

Rappelons que la relation entre types est purement syntaxique et ne capture pas la sémantique de la conformité. La vérification de la sémantique est le but des contrats comportementaux.

Contrats comportementaux

Les contrats comportementaux sont fondés sur une méthode proposée dans [Hoare 1969] pour prouver des propriétés de programmes, en utilisant des pré- et post-conditions avec des règles de preuve fondées sur la logique du premier ordre. Soit A une action séquentielle. Alors la notation

$$\{P\}\ A\ \{Q\},$$

dans lequel P et Q sont des assertions (prédicats sur l'état de l'univers du programme), a le sens suivant : si l'exécution de A est lancée dans un état dans lequel P est vrai, et si A se termine, alors Q est vrai à la fin de cette exécution. Une condition supplémentaire peut être spécifiée sous la forme d'un prédicat invariant I qui doit être preservé par l'exécution de A. Ainsi si P et I sont initialement vrais, Q et I sont vrais à la fin de A, si A se termine. L'invariant peut être utilisé pour spécifier une contrainte de cohérence.

Ceci peut être transposé comme suit en termes de services et de contrats. Avant l'exécution d'un service,

- le demandeur doit garantir la précondition P et l'invariant I,
- le fournisseur doit garantir que le service est effectivement delivré dans un temps fini, et doit assurer la postcondition Q et l'invariant I.

Les cas possibles de terminaison anormale doivent être spécifiés dans le contrat et traités par réessai ou par la levée d'une exception. Cette méthode a été développée sous le nom de « conception par contrat » [Meyer 1992] via des extensions au langage Eiffel permettant l'expression de pré- et post-conditions et de prédicats invariants. Ces conditions sont vérifiées à l'exécution. Des outils analogues ont été développés pour Java [Kramer 1998].

La notion de sous-typage peut être étendue aux contrats comportementaux, en spécifiant les contraintes de conformité pour les assertions. Soit une procédure *Proc1* définie dans l'interface *I1*, et la procédure correspondante (conforme) *Proc2* définie dans l'interface *I2*, telle que $T(I2) \sqsubseteq T(I1)$. Soit $P1$ et $Q1$ (resp. $P2$ et $Q2$) les pré- et post-conditions définies pour *Proc1* (resp. *Proc2*). Les conditions suivantes doivent être vérifiées :

$$P1 \Rightarrow P2 \text{ et } Q2 \Rightarrow Q1$$

En d'autres termes, un sous-type a des préconditions plus faibles et des postconditions plus fortes que son super-type, ce qui illustre de nouveau la condition de substitution.

Contrats de synchronisation

L'expression de la validité des programmes au moyen d'assertions peut être étendue aux programmes concurrents. Le but ici est de séparer, autant que possible, la description des contraintes de synchronisation du code des procédures. Les expressions de chemin (*path expressions*), qui spécifient des contraintes sur l'ordre et la concurrence de l'exécution des procédures, ont été proposées dans [Campbell and Habermann 1974]. Les développements ultérieurs (compteurs et politiques de synchronisation) ont essentiellement été des extensions et des raffinements de cette construction, dont la réalisation repose sur l'exécution de procédures engendrées à partir de la description statique des contraintes. Plusieurs articles décrivant des propositions dans ce domaine figurent dans [CACM 1993], mais ces techniques n'ont pas trouvé une large application.

Une forme très simple de contrat de synchronisation est la clause `synchronized` de Java, qui spécifie une exécution en exclusion mutuelle. Un autre exemple est le choix d'une politique de gestion de file d'attente (par exemple FIFO, priorité, etc.) parmi un ensemble prédéfini pour la gestion d'une resource partagée.

Les travaux plus récents (voir par exemple [Chakrabarti et al. 2002]) visent à vérifier les contraintes de synchronisation à la compilation, pour détecter assez tôt les incompatibilités.

Contrats de Qualité de Service

Les spécifications associées à l'interface d'un système ou d'une partie de système, exprimés ou non de manière formelle, sont appelés *fonctionnelles*. Un système peut en outre être l'objet de spécifications supplémentaires, qui s'appliquent à des aspects qui n'apparaissent pas explicitement dans son interface. Ces spécifications sont dites *extra-fonctionnelles*[5].

La qualité de service (un autre nom pour ces propriétés) inclut les aspects suivants.

- *Disponibilité*. La disponibilité d'un service est une mesure statistique de la fraction du temps pendant laquelle le service est prêt à être rendu. Elle dépend à la fois du taux de défaillances du système qui fournit le service et du temps nécessaire pour restaurer le service après une défaillance.
- *Performances*. Cette qualité couvre plusieurs aspects, qui sont essentiels pour les applications en temps réel (applications dont la validité ou l'utilité repose sur des contraintes temporelles). Certains de ces aspects sont liés à la communication (bornes sur la latence, la gigue, la bande passante) ; d'autres s'appliquent à la vitesse de traitement ou à la latence d'accès aux données.
- *Securité*. La sécurité couvre des propriétés liées à l'usage correct d'un service par ses utilisateurs selon des règles d'usage spécifiées. Elle comprend la confidentialité, l'intégrité, l'authentification, et le contrôle des droits d'accès.

[5]Noter que la définition d'une spécification comme « fonctionnelle » ou « extra-fonctionnelle » n'est pas absolue, mais dépend de l'état de l'art : un aspect qui est extra-fonctionnel aujourd'hui deviendra fonctionnel lorsque des progrès techniques permettront d'intégrer ses spécifications dans celles de l'interface.

D'autres aspects extra-fonctionnels, plus difficiles à quantifier, sont la maintenabilité et la facilité d'évolution.

La plupart des aspects de qualité de service étant liés à un environnement variable, il est important que les politiques de gestion de la QoS puissent être adaptables. Les contrats de QoS comportent donc généralement la possibilité de négociation, c'est-à-dire de redéfinition des termes du contrat via des échanges, à l'exécution, entre le demandeur et le fournisseur du service.

2.2 Patrons architecturaux

Dans cette section, nous examinons quelques principes de base pour la structuration des systèmes intergiciels. La plupart des systèmes examinés dans ce livre sont organisés selon ces principes, qui fournissent essentiellement des indications pour décomposer un système complexe en parties.

2.2.1 Architectures multiniveaux

Architectures en couches

La décomposition d'un système complexe en niveaux d'abstraction est un ancien et puissant principe d'organisation. Il régit beaucoup de domaines de la conception de systèmes, via des notions largement utilisées telles que les machines virtuelles et les piles de protocoles.

L'abstraction est une démarche de conception visant à construire une vue simplifiée d'un système sous la forme d'un ensemble organisé d'interfaces, qui ne rendent visibles que les aspects jugés pertinents. La réalisation de ces interfaces en termes d'entités plus détaillées est laissée à une étape ultérieure de raffinement. Un système complexe peut ainsi être décrit à différents niveaux d'abstraction. L'organisation la plus simple (Figure 2.4a) est une hiérarchie de couches, dont chaque niveau i définit ses propres entités, qui fournissent une interface au niveau supérieur (i+1). Ces entités sont réalisées en utilisant l'interface fournie par le niveau inférieur(i-1), jusqu'à un niveau de base prédéfini (généralement réalisé par matériel). Cette architecture est décrite dans [Buschmann et al. 1995] sous le nom de patron Layers.

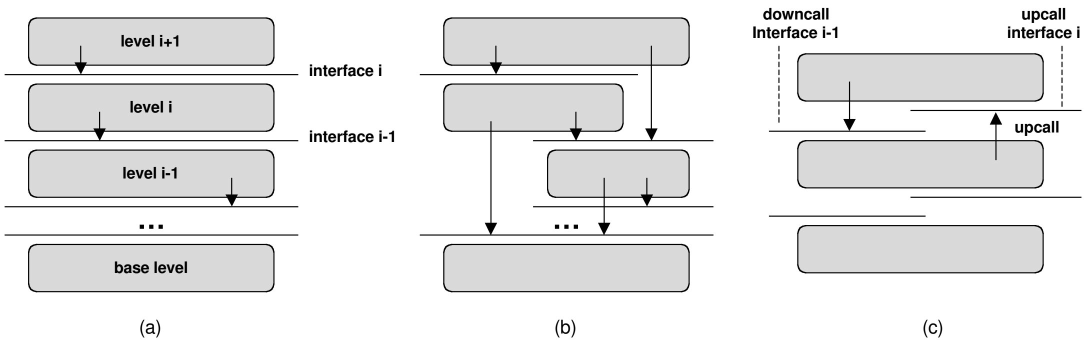

Figure 2.4 – Organisations de systèmes en couches

L'interface fournie par chaque niveau peut être vue comme un ensemble de fonctions définissant une bibliothèque, auquel cas elle est souvent appelée API (*Application Programming Interface*)[6]. Une vue alternative est de considérer chaque niveau comme une machine virtuelle, dont le « langage » (le jeu d'instructions) est défini par son interface. En vertu du principe d'encapsulation, une machine virtuelle masque les détails de réalisation de tous les niveaux inférieurs. Les machines virtuelles ont été utilisées pour émuler un ordinateur ou un système d'exploitation au-dessus d'un autre, pour émuler un nombre quelconque de ressources identiques par multiplexage d'une ressource physique, ou pour réaliser l'environnement d'exécution d'un langage de programmation (par exemple la *Java Virtual Machine* (JVM) [Lindholm and Yellin 1996]).

Ce schéma de base peut être étendu de plusieurs manières. Dans la première extension (Figure 2.4b), une couche de niveau i peut utiliser tout ou partie des interfaces fournies par les machines de niveau inférieur. Dans la seconde extension, une couche de niveau i peut rappeler la couche de niveau $i+1$, en utilisant une interface de rappel (*callback*) fournie par cette couche. Dans ce contexte, le rappel est appelé « appel ascendant » (*upcall*) (par référence à la hiérarchie « verticale » des couches).

Bien que les appels ascendants puissent être synchrones, leur utilisation la plus fréquente est la propagation d'événements asynchrones vers le haut de la hiérarchie des couches. Considérons la structure d'un noyau de système d'exploitation. La couche supérieure (application) active le noyau par appels descendants synchrones, en utilisant l'API des appels système. Le noyau active aussi les fonctions réalisées par le matériel (par exemple mettre à jour la MMU, envoyer une commande à un disque) par l'équivalent d'appels synchrones. En sens inverse, le matériel active typiquement le noyau via des interruptions asynchrones (appels ascendants), qui déclenchent l'exécution de traitants. Cette structure d'appel est souvent répétée aux niveaux plus élevés : chaque couche reçoit des appels synchrones de la couche supérieure et des appels asynchrones de la couche inférieure. Ce patron, décrit dans [Schmidt et al. 2000] sous le nom de Half Sync, Half Async, est largement utilisé dans les protocoles de communication.

Architectures multiétages

Le développement des systèmes répartis a promu une forme différente d'architecture multiniveaux. Considérons l'évolution historique d'une forme usuelle d'applications client-serveur, dans laquelle les demandes d'un client sont traitées en utilisant l'information stockée dans une base de données.

Dans les années 1970 (Figure 2.5a), les fonctions de gestion de données et l'application elle-même sont exécutées sur un serveur central (*mainframe*). Le poste du client est un simple terminal, qui réalise une forme primitive d'interface utilisateur.

Dans les années 1980 (Figure 2.5b), les stations de travail apparaissent comme machines clientes, et permettent de réaliser des interfaces graphique élaborées pour l'utilisateur. Les capacités de traitement de la station cliente lui permettent en outre de participer au traitement de l'application, reduisant ainsi la charge du serveur et améliorant la capacité de croissance (car l'addition d'une nouvelle station cliente ajoute de la puissance de traitement

[6] Une interface complexe peut aussi être partitionnée en plusieurs APIs, chacune étant liée à une fonction spécifique.

pour les applications).

L'inconvénient de cette architecture est que l'application est maintenant à cheval sur les machines client et serveur ; l'interface de communication est à présent interne à l'application. Une modification de cette dernière peut maintenant impliquer des changements à la fois sur les machines client et serveur, et éventuellement une modification de l'interface de communication.

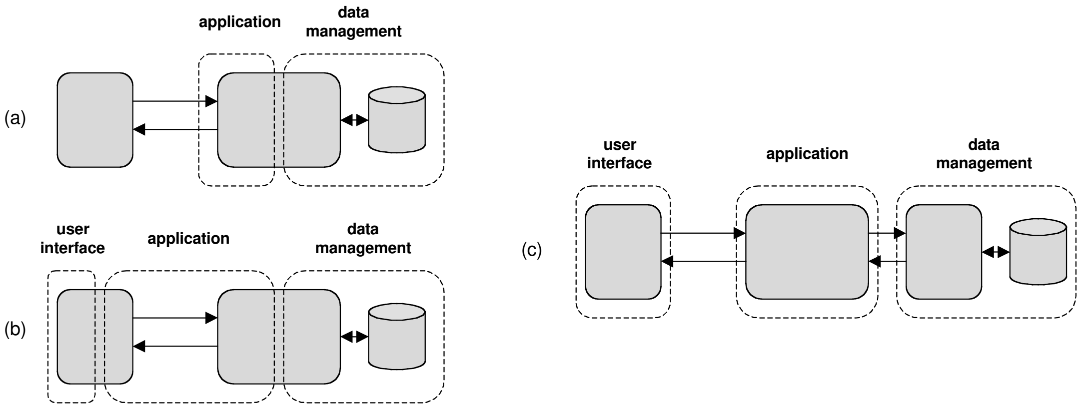

Figure 2.5 – Architectures multiétages

Ces défauts sont corrigés par l'architecture décrite sur la Figure 2.5c, introduite à la fin des années 1990. Les fonctions de l'application sont partagées entre trois machines : la station client ne réalise que l'interface graphique, l'application proprement dite réside sur un serveur dedié, et la gestion de la base de données est dévolue à une autre machine. Chacune de ces divisions « horizontales » est appelée un *étage* (en anglais *tier*). Une spécialisation plus fine des fonctions donne lieu à d'autres architectures multiétages. Noter que chaque étage peut lui-même faire l'objet d'une décomposition « verticale » en niveaux d'abstraction.

L'architecture multiétages conserve l'avantage du passage à grande échelle, à condition que les serveurs puissent être renforcés de manière incrémentale (par exemple en ajoutant des machines à une grappe). En outre les interfaces entre étages peuvent être conçues pour favoriser la séparation de préoccupations, puisque les interfaces logiques coïncident maintenant avec les interfaces de communication. Par exemple, l'interface entre l'étage d'application et l'étage de gestion de données peut être rendue générique, pour accepter facilement un nouveau type de base de données, ou pour intégrer une application patrimoniale, en utilisant un adaptateur (section 2.3.4) pour la conversion d'interface.

Des exemples d'architectures multiétages sont présentés dans le chapitre 5.

Canevas

Un canevas logiciel (en anglais *framework*) est un squelette de programme qui peut être directement réutilisé, ou adapté selon des règles bien définies, pour résoudre une famille de problèmes apparentés. Cette définition recouvre de nombreux cas d'espèce ; nous nous intéressons ici à une forme particulière de canevas composée d'une infrastructure dans laquelle des composants logiciels peuvent être insérés en vue de fournir des services spécifiques. Ces canevas illustrent des notions relatives aux interfaces, aux rappels et à l'inversion du contrôle.

Le premier exemple (Figure 2.6a) est le micronoyau, une architecture introduite dans les années 1980 et visant à développer des systèmes d'exploitation facilement configurables. Un système d'exploitation à micronoyau se compose de deux couches :

- Le micronoyau proprement dit, qui gère les resources matérielles (processeurs, mémoire, entrées-sorties, interface de réseau), et fournit au niveau supérieur une API abstraite de gestion de ressources.
- Le noyau, qui réalise un système d'exploitation spécifique (une « personnalité ») en utilisant l'API du micronoyau.

Un noyau de système d'exploitation construit sur un micronoyau est généralement organisé comme un ensemble de *serveurs*, dont chacun est chargé d'une fonction spécifique (gestion de processus, système de fichiers, etc.). Un appel système typique émis par une application est traité comme suit .

- Le noyau analyse l'appel et active le micronoyau en utilisant la fonction appropriée de son API.
- Le micronoyau rappelle un serveur dans le noyau. Au retour de cet appel ascendant, le micronoyau peut interagir avec le matériel ; cette séquence peut être itérée, par exemple si plusieurs serveurs sont en jeu.
- Le micronoyau rend la main au noyau, qui termine le travail et revient à l'application.

Pour ajouter une nouvelle fonction à un noyau, il faut donc développer et intégrer un nouveau serveur.

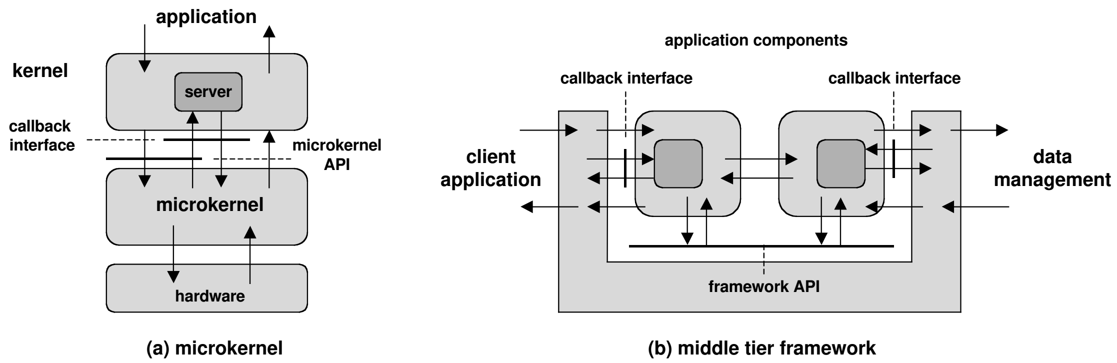

Figure 2.6 – Architectures de canevas

Le second exemple (Figure 2.6b) illustre l'organisation typique de l'étage médian d'une architecture client-serveur à 3 étages. Ce canevas interagit avec l'étage client et avec l'étage de gestion de données, et sert de médiateur pour l'interaction entre ces étages et le programme de l'application proprement dite. Ce programme est organisé comme un ensemble de composants, qui utilisent l'API fournie par le canevas et doivent fournir un ensemble d'interfaces de rappel. Ainsi une requête d'un client est traitée par le canevas, qui active les composant applicatifs appropriés, interagit avec eux en utilisant ses propres API et l'interface de rappel des composants, et retourne finalement au client.

Des exemples détaillés de cette organisation sont présentés au chapitre 5.

Les deux exemples ci-dessus illustrent l'inversion du contrôle. Pour fournir ses services, le canevas utilise des rappels vers les modules logiciels externes (serveurs dans l'exemple micronoyau, ou composants applicatifs dans l'étage médian). Ces modules doivent respecter le contrat du canevas, en fournissant des interfaces de rappel spécifiées et en utilisant l'API du canevas.

Les organisations en couches et en étages définissent une structure à gros grain pour un système complexe. L'organisation interne de chaque couche ou étage (ou couche dans un étage) utilise elle-même des entités de grain plus fin. Les objets, un moyen usuel de définir cette structure fine, sont présentés dans la section suivante.

2.2.2 Objets répartis

Programmation par objets

Les objets ont été introduits dans les années 1960 comme un moyen de structuration des systèmes logiciels. Il existe de nombreuses définitions des objets, mais les propriétés suivantes en capturent les concepts plus courants, particulièrement dans le contexte de la programmation répartie.

Un *objet*, dans un modèle de programmation, est une représentation logicielle d'une entité du monde réel (telle qu'une personne, un compte bancaire, un document, une voiture, etc.). Un objet est l'association d'un état et d'un ensemble de procédures (ou méthodes) qui opèrent sur cet état. Le modèle d'objets que nous considérons a les propriétés suivantes.

– *Encapsulation*. Un objet a une interface, qui comprend un ensemble de méthodes (procédures) et d'attributs (valeurs qui peuvent être lues et modifiées). La seule manière d'accéder à un objet (pour consulter ou modifier son état) est d'utiliser son interface. Les seules parties de l'état visibles depuis l'extérieur de l'objet sont celles explicitement présentes dans l'interface ; l'utilisation d'un objet ne doit reposer sur aucune hypothèse sur sa réalisation. Le type d'un objet est défini par son interface. Comme indiqué en 2.1.2, l'encapsulation assure l'indépendance entre interface et réalisation. L'interface joue le rôle d'un contrat entre l'utilisateur et le réalisateur d'un objet. Un changement dans la réalisation d'un objet est fonctionnellement invisible à ses utilisateurs, tant que l'interface est préservée.
– *Classes et instances*. Une *classe* est une description générique commune à un ensemble d'objets (les instances de la classe). Les instances d'une classe ont la même interface (donc le même type), et leur état a la même structure ; mais chaque instance a son propre exemplaire de l'état, et elle est identifiée comme une entité distincte. Les instances d'une classe sont créées dynamiquement, par une opération appelée *instanciation* ; elles peuvent aussi être dynamiquement détruites, soit explicitement soit automatiquement (par un ramasse-miettes) selon la réalisation spécifique du modèle d'objet.
– *Héritage*. Une classe peut dériver d'une autre classe par spécialisation, autrement dit par définition de méthodes et/ou d'attributs supplémentaires, ou par redéfinition (surcharge) de méthodes existantes. On dit que la classe dérivée *étend* la classe initiale (ou classe de base) ou qu'elle *hérite* de cette classe. Certains modèles permettent à une classe d'hériter de plus d'une classe (héritage multiple).
– *Polymorphisme*. Le polymorphisme est la capacité, pour une méthode, d'accepter des

paramètres de différents types et d'avoir un comportement différent pour chacun de ces types. Ainsi un objet peut être remplacé, comme paramètre d'une méthode, par un objet « compatible ». La notion de compatibilité, ou conformité (voir section 2.1.3) est exprimée par une relation entre types, qui dépend du modèle spécifique de programmation ou du langage utilisé.

Rappelons que ces définitions ne sont pas universelles, et ne sont pas applicables à tous les modèles d'objets (par exemple il y a d'autres mécanismes que les classes pour créer des instances, les objets peuvent être actifs, etc.), mais elles sont représentatives d'un vaste ensemble de modèles utilisés dans la pratique, et sont mises en œuvre dans des langages tels que Smalltalk, C++, Eiffel, Java, ou C#.

Objets distants

Les propriétés ci-dessus font que les objets sont un bon mécanisme de structuration pour les systèmes répartis.

– L'hétérogénéité est un trait dominant de ces systèmes. L'encapsulation est un outil puissant dans un environnement hétérogène : l'utilisateur d'un objet doit seulement connaître une interface pour cet objet, qui peut avoir des réalisations différentes sur différents sites.
– La création dynamique d'instances d'objets permet de construire un ensemble d'objets ayant la même interface, éventuellement sur des sites distants différents ; dans ce cas l'intergiciel doit fournir un mécanisme pour la création d'objets distants, sous la forme de fabriques (voir section 2.3.2).
– L'héritage est un mécanisme de réutilisation, car il permet de définir une nouvelle interface à partir d'une interface existante. Il est donc utile pour les développeurs d'applications réparties, qui travaillent dans un environnement changeant et doivent définir de nouvelles classes pour traiter des nouvelles situations. Pour utiliser l'héritage, on conçoit d'abord une classe (de base) générique pour capturer un ensemble de traits communs à une large gamme de situations attendues. Des classes spécifiques, plus spécialisées, sont alors définies par extension de la classe de base. Par exemple, une interface pour un flot vidéo en couleur peut être définie comme une extension de celle d'un flot vidéo générique. Une application qui utilise des flots d'objets vidéo accepte aussi des flots d'objets en couleur, puisque ces objets réalisent l'interface des flots vidéo (c'est un exemple de polymorphisme).

La manière la plus simple et la plus courante pour répartir des objets est de permettre aux objets qui constituent une application d'être situés sur un ensemble de sites répartis (autrement dit, l'objet est l'unité de répartition ; d'autres méthodes permettent de partitionner la représentation d'un objet entre plusieurs sites). Une application cliente peut utiliser un objet situé sur un site distant en appelant une méthode de l'interface de l'objet, comme si l'objet était local. Des objets utilisés de cette manière sont appelés *objets distants*, et leur mode d'interaction est l'appel d'objet distant (*Remote Method Invocation*) ; c'est la transposition du RPC au monde des objets.

Les objets distants sont un exemple d'une organisation client-serveur. Comme un client peut utiliser plusieurs objets différents situés sur un même site distant, des termes distincts sont utilisés pour désigner le site distant (le site *serveur*) et un objet individuel qui fournit un service spécifique (un objet *servant*). Pour que le système fonctionne, un intergiciel approprié doit localiser une réalisation de l'objet servant sur un site éventuellement distant, envoyer les paramètres sur l'emplacement de l'objet, réaliser l'appel effectif, et renvoyer les résultats à l'appelant. Un intergiciel qui réalise ces fonctions est un courtier d'objets répartis (en anglais *Object Request Broker*, ou ORB).

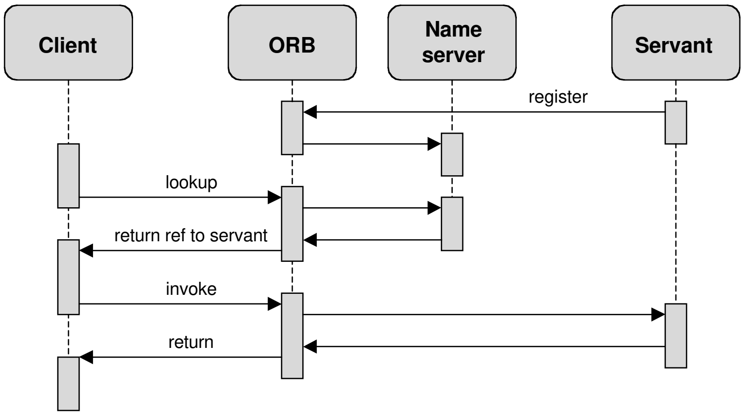

Figure 2.7 – Appel de méthode à distance

La structure d'ensemble d'un appel à un objet distant (Figure 2.7) est semblable à celle d'un RPC : l'objet distant doit d'abord être localisé, ce qui est généralement fait au moyen d'un serveur des noms ou d'un service vendeur (*trader*) ; l'appel proprement dit est ensuite réalisé. L'ORB sert de médiateur aussi bien pour la recherche que pour l'appel.

2.3 Patrons pour l'intergiciel à objets répartis

Les mécanismes d'exécution à distance reposent sur quelques patrons de conception qui ont été largement décrits dans la littérature, en particulier dans [Gamma et al. 1994], [Buschmann et al. 1995], et [Schmidt et al. 2000]. Dans cette présentation, nous mettons l'accent sur l'utilisation spécifique de ces patrons pour l'intergiciel réparti à objets, et nous examinons leurs similitudes et leurs différences. Pour une discussion plus approfondie de ces patrons, voir les références indiquées.

2.3.1 *Proxy*

Proxy (ce terme anglais est traduit par « représentant » ou « mandataire ») est un des premiers patrons de conception identifiés en programmation répartie [Shapiro 1986, Buschmann et al. 1995]. Nous n'examinons ici que son utilisation pour les objets répartis, bien que son domaine d'application ait été étendu à de nombreuses autres constructions.

1. **Contexte**. Le patron Proxy est utilisé pour des applications organisées comme un ensemble d'objets dans un environnement réparti, communicant au moyen d'appels de méthode à distance : un client demande un service fourni par un objet éventuellement distant (le servant).

2. **Problème**. Définir un mécanisme d'accès qui n'implique pas de coder « en dur » l'emplacement du servant dans le code client, et qui ne nécessite pas une connaissance détaillée des protocoles de communication par le client.
3. **Propriétés souhaitées**. L'accès doit être efficace à l'exécution. La programmation doit être simple pour le client ; idéalement, il ne doit pas y avoir de différence entre accès local et accès distant (cette propriété est appelée transparence d'accès).
4. **Contraintes**. La principale contrainte résulte de l'environnement réparti : le client et le serveur sont dans des espaces d'adressage différents.
5. **Solution**. Utiliser un représentant local du serveur sur le site du client. Ce représentant, ou mandataire, a exactement la même interface que le servant. Toute l'information relative au système de communication et à la localisation du servant est cachée dans le mandataire, et ainsi rendue invisible au client.

   L'organisation d'un mandataire est illustrée sur la figure 2.8.

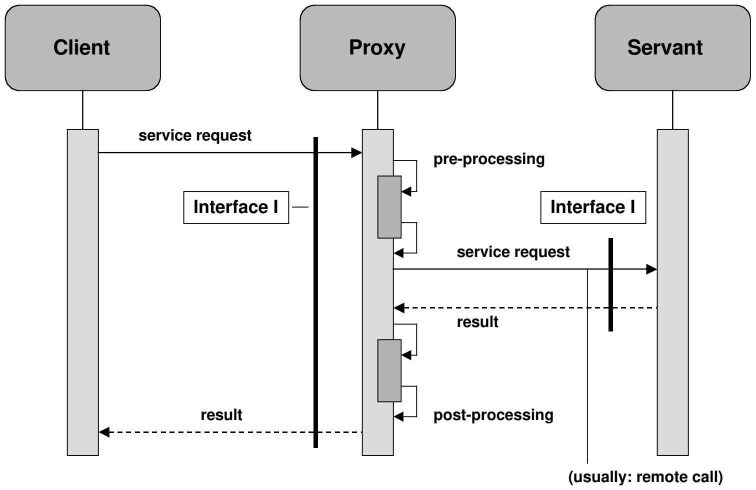

Figure 2.8 – Le patron PROXY

   La structure interne d'un mandataire suit un schéma bien défini, qui facilite sa génération automatique.
   - une phase de pré-traitement, qui consiste essentiellement à emballer les paramètres et à préparer le message de requête,
   - l'appel effectif du servant, utilisant le protocole de communication sous-jacent pour envoyer la requête et pour recevoir la réponse,
   - une phase de post-traitement, qui consiste essentiellement à déballer les valeurs de retour.
6. **Usages connus**.

   Dans la construction de l'intergiciel, les mandataires sont utilisés comme représentants locaux pour des objets distants. Ils n'ajoutent aucune fonction. C'est le cas des souches (*stubs*) et des squelettes utilisés dans RPC ou Java-RMI.

   Des variantes des *proxies* contiennent des fonctions supplémentaires. Des exemples en sont les caches et l'adaptation côté client. Dans ce dernier cas, le *proxy* peut filtrer la sortie du serveur pour l'adapter aux capacités spécifiques d'affichage du client (couleur, résolution, etc.). De tels mandataires « intelligents » (*smart proxies*) combinent les fonctions standard d'un mandataire avec celles d'un intercepteur (voir section 2.3.5).

7. **Références**.

   Une discussion du patron PROXY peut être trouvée dans [Gamma et al. 1994], [Buschmann et al. 1995].

2.3.2 *Factory*

1. **Contexte**. On considère des applications organisées comme un ensemble d'objets dans un environnement réparti (la notion d'objet dans ce contexte peut être très générale, et n'est pas limitée au domaine strict de la programmation par objets).
2. **Problème**. On souhaite pouvoir créer dynamiquement des familles d'objets apparentés (par exemple des instances d'une même classe), tout en permettant de reporter certaines décisions jusqu'à la phase d'exécution (par exemple le choix d'une classe concrète pour réaliser une interface donnée).
3. **Propriétés souhaitées**. Les détails de réalisation des objets créés doivent être invisibles. Le processus de création doit pouvoir être paramétré. L'évolution du mécanisme doit être facilitée (pas de décision « en dur »).
4. **Contraintes**. La principale contrainte résulte de l'environnement réparti : le client (qui demande la création de l'objet) et le serveur (qui crée effectivement l'objet) sont dans des espaces d'adressage différents.
5. **Solution**. Utiliser deux patrons corrélés : une usine abstraite ABSTRACT FACTORY définit une interface et une organisation génériques pour la création d'objets ; la création est déléguée à des usines concrètes. ABSTRACT FACTORY peut être réalisé en utilisant FACTORY METHODS (une méthode de création redéfinie dans une sous-classe).

   Un autre manière d'améliorer la souplesse est d'utiliser une usine de fabrication d'usines, comme illustré sur la Figure 2.9 (le mécanisme de création est lui-même paramétré).

   Un usine peut aussi être utilisée comme un gestionnaire des objets qu'elle a créés, et peut ainsi réaliser une méthode pour localiser un objet (en renvoyant une référence pour cet objet), et pour détruire un objet sur demande.
6. **Usages connus**.

   FACTORY est un des patrons les plus largement utilisés dans l'intergiciel. Il sert à la fois dans des applications (pour créer des instances distantes d'objets applicatifs) et dans l'intergiciel lui-même (un exemple courant est l'usine à liaisons). Les usines sont aussi utilisées en liaison avec les composants (voir chapitres 5 et 7).
7. **Références**.

   Les deux patrons ABSTRACT FACTORY et FACTORY METHOD sont décrits dans [Gamma et al. 1994].

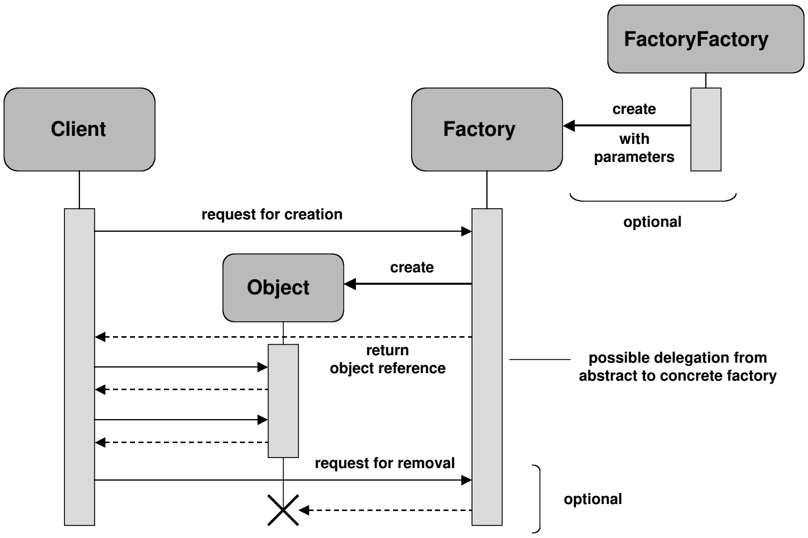

Figure 2.9 – Le patron FACTORY

2.3.3 *Pool*

Le patron POOL est un complément à FACTORY, qui vise à réduire le temps d'exécution de la création et de la destruction d'objets, en construisant à l'avance (lors d'une phase d'initialisation) une réserve (*pool*) d'objets. Cela se justifie si le coût de la création et de la destruction est élevé par rapport à celui des opérations sur la réserve. Les opérations de création et de destruction deviennent alors :

```
Obj create(params) {                remove(Obj obj) {
   if (pool empty)                    if (pool full)
      obj = new Obj                      delete(obj)
       /* utilise Factory */          else {
   else                                  obj.cleanup()
      obj = pool.get()                   pool.put(obj)}
   obj.init(params)                   }
   return (obj)
   }
```

Les opérations `init` et `cleanup` permettent respectivement, si nécessaire, d'initialiser l'état de l'objet créé et de remettre l'objet dans un état neutre.

On a supposé ici que la taille de la réserve était fixe. Il est possible d'ajuster la taille du pool en fonction de la demande observée. On peut encore raffiner le fonctionnement en maintenant le nombre d'objets dans la réserve au-dessus d'un certain seuil, en déclenchant les créations nécessaires si ce nombre d'objets tombe au-dessous du seuil. Cette régulation peut éventuellement se faire en travail de fond pour profiter des temps libres.

Trois cas fréquents d'usage de ce patron sont :

– La gestion de la mémoire. Dans ce cas, on peut prévoir plusieurs réserves de zones préallouées de tailles différentes, pour répondre aux demandes le plus fréquemment

observées.
- La gestion des *threads* ou des processus.
- La gestion des composants dans certains canevas (par exemple les *Entity Beans* dans la plate-forme EJB, voir chapitre 5)

Dans tous ces cas, le coût élevé de création des entités justifie largement l'usage d'une réserve.

2.3.4 *Adapter*

1. **Contexte**. Le contexte est celui de la fourniture de services, dans un environnement réparti : un service est défini par une interface ; les clients demandent des services ; des servants, situés sur des serveurs distants, fournissent des services.
2. **Problème**. On souhaite réutiliser un servant existant en le dotant d'une nouvelle interface conforme à celle attendue par un client (ou une classe de clients).
3. **Propriétés souhaitées**. Le mécanisme de conversion d'interface doit être efficace à l'exécution. Il doit aussi être facilement adaptable, pour répondre à des changements imprévus des besoins. Il doit être réutilisable (c'est-à-dire générique).
4. **Contraintes**. Pas de contraintes spécifiques.
5. **Solution**. Fournir un composant (l'adaptateur, ou *wrapper*) qui isole le servant en interceptant les appels de méthode à son interface. Chaque appel est précédé par un prologue et suivi par un épilogue dans l'adaptateur (Figure 2.10). Les paramètres et résultats peuvent nécessiter une conversion.

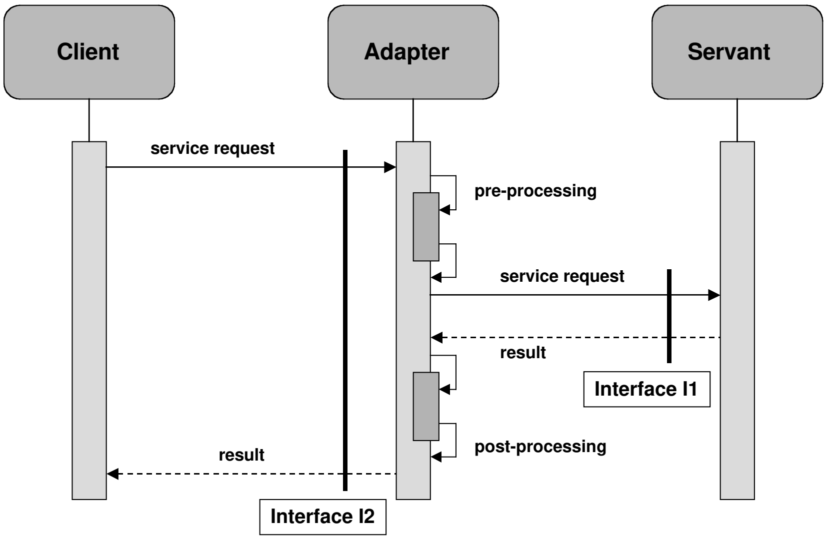

Figure 2.10 – Le patron ADAPTER

Dans des cas simples, un adaptateur peut être automatiquement engendré à partir d'une description des interfaces fournie et requise.

6. **Usages connus**.

Les adaptateurs sont largement utilisés dans l'intergiciel pour encapsuler des fonctions côté serveur. Des exemples sont le *Portable Objet Adapter* (POA) de CORBA et les divers adaptateurs pour la réutilisation de logiciel patrimoniaux (*legacy systems*), tel que *Java Connector Architecture* (JCA).

7. **Références**.

   Adapter (également appelé Wrapper) est décrit dans [Gamma et al. 1994]. Un patron apparenté est Wrapper Façade ([Schmidt et al. 2000]), qui fournit une interface de haut niveau (par exemple sous forme d'objet) à des fonctions de bas niveau.

2.3.5 *Interceptor*

1. **Contexte**. Le contexte est celui de la fourniture de services, dans un environnement réparti : un service est défini par une interface ; les clients demandent des services ; les servants, situés sur des serveurs distants, fournissent des services. Il n'y a pas de restrictions sur la forme de la communication (uni- or bi-directionnelle, synchrone ou asynchrone, etc.).
2. **Problème**. On veut ajouter de nouvelles capacités à un service existant, ou fournir le service par un moyen différent.
3. **Propriétés souhaitées**. Le mécanisme doit être générique (applicable à une large variété de situations). Il doit permettre de modifier un service aussi bien statiquement (à la compilation) que dynamiquement (à l'exécution).
4. **Contraintes**. Les services peuvent être ajoutés ou supprimés dynamiquement.
5. **Solution**. Créer (statiquement ou dynamiquement) des objets d'interposition, ou intercepteurs. Ces objets interceptent les appels (et/ou les retours) et insérent un traitement spécifique, qui peut être fondé sur une analyse du contenu. Un intercepteur peut aussi rediriger un appel vers une cible différente.

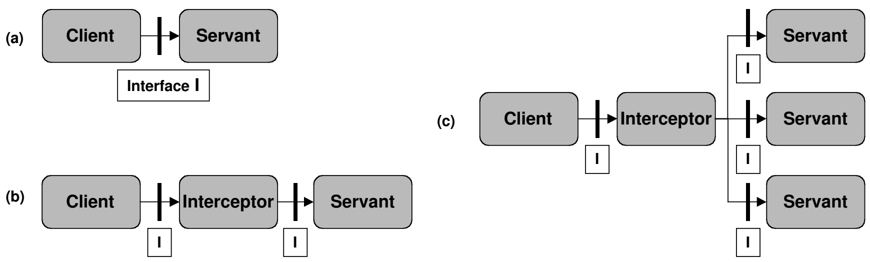

Figure 2.11 – Formes simples d'intercepteur

Ce mécanisme peut être réalisé sous différentes formes. Dans la forme la plus simple, un intercepteur est un module qui est inseré à un point spécifié dans le chemin d'appel entre le demandeur et le fournisseur d'un service (Figure 2.11a et 2.11b). Il peut aussi être utilisé comme un aiguillage entre plusieurs servants qui peuvent fournir le même service avec différentes options (Figure 2.11c), par exemple l'ajout de fonctions de tolérance aux fautes, d'équilibrage de charge ou de caches.

Sous une forme plus générale (Figure 2.12), intercepteurs et fournisseurs de service (servants) sont gérés par une infrastructure commune et créés sur demande. L'intercepteur utilise l'interface du servant et peut aussi s'appuyer sur des services fournis par l'infrastructure. Le servant peut fournir des fonctions de rappel utilisables par l'intercepteur.

6. **Usages connus**.
   Les intercepteurs sont utilisés dans une grande variété de situations dans les systèmes intergiciels.
   - pour ajouter des nouvelles capacités à des applications ou systèmes existants. Un exemple ancien est le mécanisme des « sous-contrats »[Hamilton et al. 1993]. Les *Portable Interceptors* de CORBA donnent une manière systématique d'étendre les fonctions du courtier d'objets (ORB) par insertion de modules d'interception en des points predéfinis dans le chemin d'appel. Un autre usage est l'aide aux mécanismes de tolérance aux fautes (par exemple la gestion de groupes d'objets).
   - pour choisir une réalisation spécifique d'un servant à l'exécution.
   - pour réaliser un canevas pour des applications à base de composants (voir chapitres 5 et 7).
   - pour réaliser un intergiciel réflexif (voir 2.4.1 and 2.4.3).

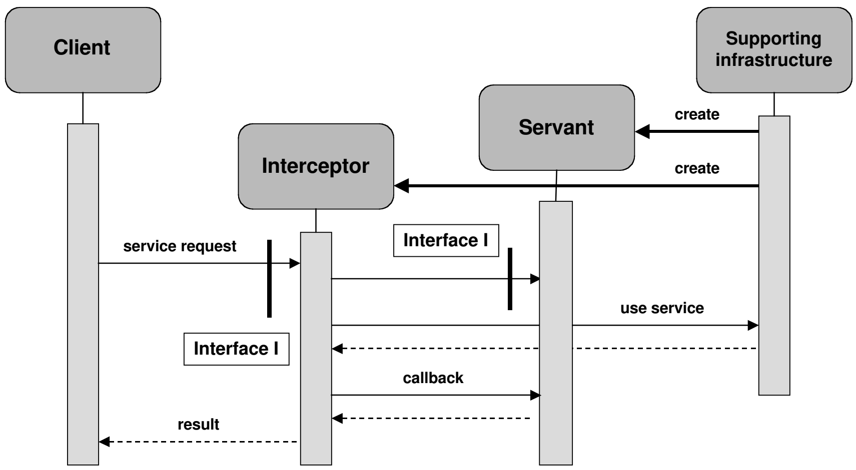

Figure 2.12 – Forme générale d'un intercepteur

7. **Références**.
   Le patron Interceptor est décrit dans [Schmidt et al. 2000].

2.3.6 Comparaison et combinaison des patrons

Trois des patrons décrits ci-dessus (Proxy, Adapter, et Interceptor) ont d'étroites relations mutuelles. Ils reposent tous trois sur un module logiciel inseré entre le demandeur et le fournisseur d'un service. Nous résumons ci-après leurs analogies et leurs différences.

- Adapter *vs* Proxy. Adapter et Proxy ont une structure semblable. Proxy préserve l'interface, alors qu'Adapter transforme l'interface. En outre, Proxy implique souvent (pas toujours) un accès à distance, alors que Adapter est généralement local.

- Adapter *vs* Interceptor. Adapter et Interceptor ont une fonction semblable : l'un et l'autre modifient un service existant. La principale différence est que Adapter transforme l'interface, alors que Interceptor transforme la fonction (de fait Interceptor peut complètement masquer la cible initiale de l'appel, en la remplaçant par un servant différent).
- Proxy *vs* Interceptor. Un Proxy peut être vu comme une forme spéciale d'un Interceptor, dont la fonction se réduit à acheminer une requête vers un servant distant, en réalisant les transformations de données nécessaires à la transmission, d'une manière indépendante des protocoles de communication. En fait, comme mentionné dans 2.3.1, un *proxy* peut être combiné avec un intercepteur, devenant ainsi « intelligent » (c'est-à-dire fournissant de nouvelles fonctions en plus de la transmission des requêtes, mais laissant l'interface inchangée).

En utilisant les patrons ci-dessus, on peut tracer un premier schéma grossier et incomplet de l'organisation d'ensemble d'un ORB (Figure 2.13).

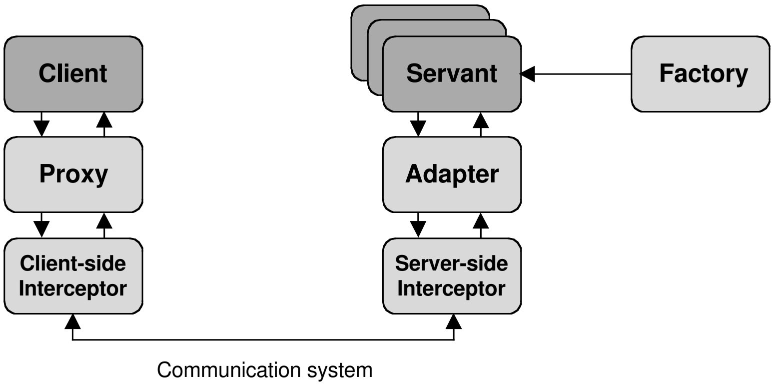

Figure 2.13 – Utilisation de patrons dans un ORB

Les principaux aspects manquants sont ceux relatifs à la liaison et à la communication.

2.4 Adaptabilité et séparation des préoccupations

Trois approches principales sont utilisées pour assurer l'adaptabilité et la séparation de préoccupations dans les systèmes intergiciels : les protocoles à méta-objets, la programmation par aspects, et les approches pragmatiques. Elles sont résumées dans les sections qui suivent.

2.4.1 Protocoles à méta-objets

La réflexion a été introduite au chapitre 1, section 1.4.2. Rappelons qu'un système réflexif est capable d'examiner et de modifier son propre comportement, en utilisant une représentation causalement connectée de lui-même.

La réflexion est une propriété intéressante pour un intergiciel, parce qu'un tel système fonctionne dans un environnement qui évolue, et doit adapter son comportement à des besoins changeants. Des capacités réflexives sont présentes dans la plupart des systèmes intergiciels existants, mais sont généralement introduites localement, pour des traits isolés. Des plates-formes intergicielles dont l'architecture de base intègre la réflexion sont développées comme prototypes de recherche [RM 2000].

Une approche générale de la conception d'un système réflexif consiste à l'organiser en deux niveaux.

– Le *niveau de base*, qui fournit les fonctions définies par les spécifications du système.
– Le *méta-niveau*, qui utilise une représentation des entités du niveau de base pour observer ou modifier le comportement de ce niveau.

Cette décomposition peut être itérée en considérant le méta-niveau comme un niveau de base pour un méta-méta-niveau, et ainsi de suite, définissant ainsi une « tour réflexive ». Dans la plupart des cas pratiques, la tour est limitée à deux ou trois niveaux.

La définition d'une représentation du niveau de base, destinée à être utilisée par le méta-niveau, est un processus appelé *réification*. Il conduit à définir des méta-objets, dont chacun est une représentation, au méta-niveau, d'une structure de données ou d'une opération définie au niveau de base. Le fonctionnement des méta-objets, et leur relation aux entités du niveau de base, sont spécifiées par un *protocole à méta-objets* (MOP) [Kiczales et al. 1991].

Un exemple simple de MOP (emprunté à [Bruneton 2001]) est la réification d'un appel de méthode dans un système réflexif à objets. Au méta-niveau, un méta-objet `Méta_Obj` est associé à chaque objet `Obj`. L'exécution d'un appel de méthode `Obj.meth(params)` comporte les étapes suivantes (Figure 2.14).

1. L'appel de méthode est réifié dans un objet `m`, qui contient une représentation de `meth` et `params`. La forme précise de cette représentation est définie par le MOP. Cet objet `m` est transmis au méta-objet, qui exécute `Méta_Obj.méta_MethodCall(m)`.

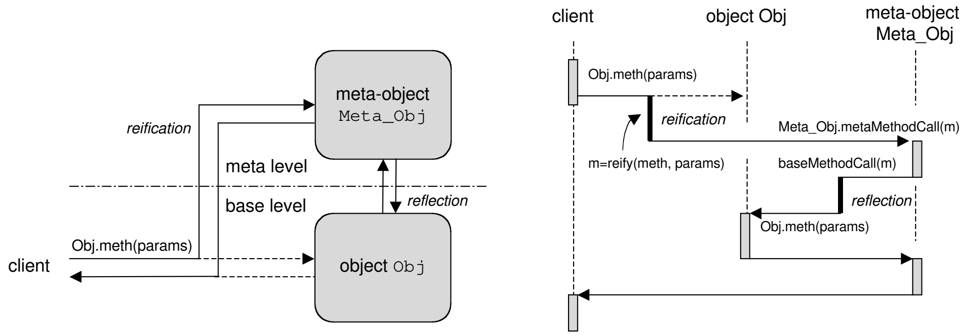

Figure 2.14 – Exécution d'un appel de méthode dans un système réflexif

2. La méthode `méta_MethodCall(m)` exécute alors les traitements spécifiés par le MOP. Pour prendre un exemple simple, il peut imprimer le nom de la méthode en vue d'une trace avant son exécution effective (en appelant une méthode telle que `m.methName.printName()`) ou il peut sauvegarder l'état de l'objet avant l'appel de

la méthode pour permettre un retour en arrière (*undo*), ou il peut vérifier la valeur des paramètres, etc.

3. Le méta-objet peut maintenant effectivement exécuter l'appel initial[7], en appelant une méthode `baseMethodCall(m)` qui exécute essentiellement `Obj.meth(params)`[8]. Cette étape (l'inverse de la réification) est appelée *réflexion*.
4. Le méta-objet exécute alors tout post-traitement défini par le MOP, et retourne à l'appelant initial.

De même, l'opération de création d'objet peut être réifiée en appelant une usine à méta-objets (au méta-niveau). Cette usine crée un objet au niveau de base, en utilisant l'usine de ce niveau ; le nouvel objet fait alors un appel ascendant (*upcall*) à l'usine à méta-objets, qui crée le méta-objet associé, et exécute toutes les opérations supplémentaires spécifiées par le MOP (Figure 2.15).

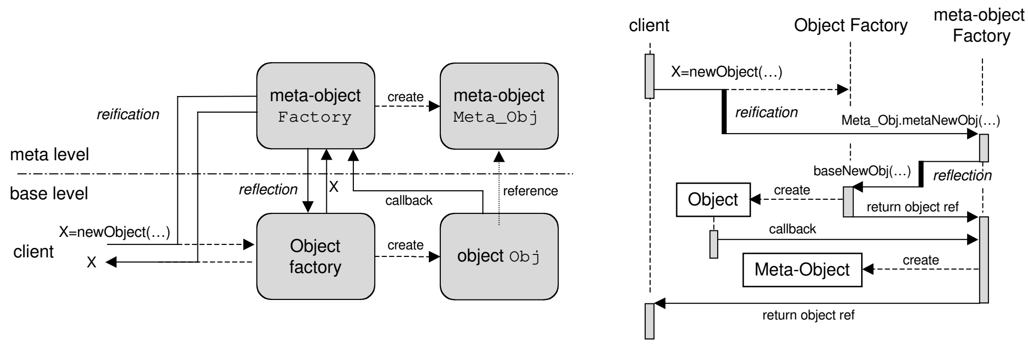

Figure 2.15 – Création d'objet dans un système réflexif

2.4.2 Programmation par aspects

La programmation par aspects (en anglais *Aspect-Oriented Programming* ou AOP) [Kiczales 1996] est motivée par les remarques suivantes.

- Beaucoup de préoccupations différentes (ou « aspects ») sont généralement présentes dans une application (des exemples usuels sont la sécurité, la persistance, la tolérance aux fautes, et d'autres aspects extra-fonctionnels).
- Le code lié à ces préoccupations est souvent étroitement imbriqué avec le code « fonctionnel » de l'application, ce qui rend les modifications et les additions difficiles et sujettes aux erreurs.

Le but de l'AOP est de définir des méthodes et outils pour mieux identifier et isoler le code relatif aux divers aspects présents dans une application. Plus précisément, une application développée avec l'AOP est construite en deux phases.

[7]Il n'exécute pas *nécessairement* l'appel initial ; par exemple, si le MOP est utilisé pour la protection, il peut décider que l'appel ne doit pas être exécuté, et revenir à l'appelant avec un message de violation de protection.

[8]Noter qu'il n'est pas possible d' appeler directement `Obj.meth(params)` parce que seule la forme réifiée de l'appel de méthode est accessible au méta-objet et aussi parce qu'une étape de post-traitement peut être nécessaire.

- La partie principale de l'application (le programme de base), et les parties qui traitent des différents aspects supplémentaires sont écrites indépendamment, en utilisant éventuellement des langages spécialisés pour le code des aspects.
- Toutes ces parties sont intégrées pour former l'application globale, en utilisant un outil de composition, le tisseur d'aspects (*aspect weaver*).

Un point de jonction (*join point*) est un emplacement, dans le code source du programme de base, où du code lié aux aspects peut être inséré. Le tissage d'aspects repose sur deux notions principales : le point de coupure (*point cut*), c'est-à-dire la spécification d'un ensemble de points de jonction selon un critère donné, et l'indication (*advice*), c'est-à-dire la définition de l'interaction du code inséré avec le code de base. Par exemple, si l'AOP est ajouté à un langage à objets, un point de coupure particulier peut être défini comme l'ensemble des points d'appel à une famille de méthodes (spécifiée par une expression regulière), ou l'ensemble des appels à un constructeur spécifié, etc. Une indication spécifie si le code inséré doit être exécuté avant, après, ou en remplacement des opérations situées aux points de coupure (dans le dernier cas, ces opérations peuvent toujours être appelées depuis le code inséré). La composition peut être faite statiquement (à la compilation), dynamiquement (à l'exécution), ou en combinant des techniques statiques et dynamiques.

Un problème important de l'AOP est la composition des aspects. Par exemple, si différents fragments de code liés aux aspects sont insérés au même point de jonction, l'ordre d'insertion peut être significatif si les aspects correspondants ne sont pas indépendants. Cette question ne peut généralement pas être décidée par le tisseur et nécessite une spécification supplémentaire.

Deux exemples d'outils qui réalisent l'AOP sont AspectJ [Kiczales et al. 2001] et JAC [Pawlak et al. 2001]. Ils s'appliquent à des programmes de base en Java.

AspectJ

AspectJ permet de définir les aspects en spécifiant ces derniers et le programme de base dans un code source Java, qui peut alors être compilé.

Un exemple simple donne une idée des capacités d'AspectJ. Le code présenté Figure 2.16 décrit un aspect, sous la forme de définition de points de coupure et d'indications.

```
public aspect MethodWrapping{

/* définition de point de coupure */
    pointcut Wrappable(): call(public * MyClass.*(..));

/* définition d'indication        */
    around(): Wrappable() {
        prelude ;   /* séquence de code devant être insérée avant un appel */
        proceed (); /* exécution d'un appel à la  méthode originelle       */
        postlude    /* séquence de code devant être insérée après un appel */
    }
}
```

Figure 2.16 – Définition d'un aspect en AspectJ.

La première partie de la description définit un point de coupure (*point cut*) comme tout appel d'une méthode publique de la classe `MyClass`. La partie indication (*advice*) indique qu'un appel à une telle méthode doit être remplacé par un prologue spécifié, suivi par un appel à la méthode d'origine, suivi par un épilogue spécifié. De fait, cela revient à placer une simple enveloppe (sans modification d'interface) autour de chaque appel de méthode spécifié dans la définition du point de coupure. Cela peut être utilisé pour ajouter des capacités de journalisation à une application existante, ou pour insérer du code de test pour évaluer des pré- et post-conditions dans une conception par contrat (2.1.3).

Une autre capacité d'AspectJ est l'*introduction*, qui permet d'insérer des déclarations et méthodes supplémentaires à des endroits spécifiés dans une classe ou une interface existante. Cette possibilité doit être utilisée avec précaution, car elle peut violer le principe d'encapsulation.

JAC

JAC (*Java Aspect Components*) a des objectifs voisins de ceux d'AspectJ. Il permet d'ajouter des capacités supplémentaires (mise sous enveloppe de méthodes, introduction) à une application existante. JAC diffère d'AspectJ sur les points suivants.

- JAC n'est pas une extension de langage, mais un canevas qui peut être utilisé à l'exécution. Ainsi des aspects peuvent être dynamiquement ajoutés à une application en cours d'exécution. JAC utilise la modification du *bytecode*, et le code des classes d'application est modifié lors du chargement des classes.
- Les *point cuts* et les *advices* sont définis séparément. La liaison entre *point cuts* et *advices* est retardée jusqu'à la phase de tissage ; elle repose sur des informations fournies dans un fichier de configuration séparé. La composition d'aspects est définie par un protocole à méta-objets.

Ainsi JAC autorise une programmation souple, mais au prix d'un surcoût à l'exécution dû au tissage dynamique des aspects dans le *bytecode*.

2.4.3 Approches pragmatiques

Les approches pragmatiques de la réflexion dans l'intergiciel s'inspirent des approches systématiques ci-dessus, mais les appliquent en général de manière ad hoc, essentiellement pour des raisons d'efficacité. Ces approches sont principalement fondées sur l'interception.

Beaucoup de systèmes intergiciels définissent un chemin d'appel depuis un client vers un serveur distant, traversant plusieurs couches (application, intergiciel, système d'exploitation, protocoles de communication). Les intercepteurs peuvent être insérés en divers points de ce chemin, par exemple à l'envoi et à la réception des requêtes et des réponses.

L'insertion d'intercepteurs permet une extension non-intrusive des fonctions d'un intergiciel, sans modifier le code des applications ou l'intergiciel lui-même. Cette technique peut être considérée comme une manière ad hoc pour réaliser l'AOP : les points d'insertion sont les points de jonction et les intercepteurs réalisent directement les aspects. En spécifiant convenablement les points d'insertion pour une classe donnée d'intergiciel, conforme à une norme spécifique (par exemple CORBA, EJB), les intercepteurs peuvent être rendus génériques et peuvent être réutilisés avec différentes réalisations de cette norme. Les fonctions qui peuvent être ajoutées ou modifiées par des intercepteurs sont notamment la surveillance (*monitoring*), la journalisation, l'enregistrement de mesures, la securité, la gestion de caches, l'équilibrage de charge, la duplication.

Cette technique peut aussi être combinée avec un protocole à méta-objets, l'intercepteur pouvant être inséré dans la partie réifiée du chemin d'appel (donc dans un méta-niveau).

Les techniques d'interception entraînent un surcoût à l'exécution. Ce coût peut être réduit par l'usage de l'*injection de code*, c'est-à-dire par intégration directe du code de l'intercepteur dans le code du client ou du serveur (c'est l'analogue de l'insertion (*inlining*) du code des procédures dans un compilateur optimisé). Pour être efficace, cette injection doit être réalisée à bas niveau, c'est-à-dire dans le langage d'assemblage, ou (pour Java) au niveau du *bytecode*, grâce à des outils appropriés tels que BCEL [BCEL], Javassist [Tatsubori et al. 2001], ou ASM [ASM 2002]. Pour préserver la souplesse d'utilisation, il doit être possible d'annuler le processus d'injection de code en revenant au format de l'interception. Un exemple d'utilisation de l'injection de code peut être trouvé dans [Hagimont and De Palma 2002].

2.4.4 Comparaison des approches

Les principales approches de la séparation de préoccupations dans l'intergiciel peuvent être comparées comme suit.

1. Les approches fondées sur les protocoles à méta-objets (MOP) sont les plus générales et les plus systématiques. Néanmoins, elles entraînent un surcoût potentiel dû au va et vient entre méta-niveau et niveau de base.
2. Les approches fondées sur les aspects (AOP) agissent à un grain plus fin que celles utilisant les MOPs et accroissent la souplesse d'utilisation, au détriment de la généralité. Les deux approches peuvent être combinéesé ; par exemple les aspects peuvet être utilisés pour modifier les opérations aussi bien au niveau de base qu'aux méta-niveaux.
3. Les approches fondées sur l'interception ont des capacités restreintes par rapport à MOP ou AOP, mais apportent des solutions acceptables dans de nombreuses situations pratiques. Elles manquent toujours d'un modèle formel pour les outils de conception et de vérification.

Dans tous les cas, des techniques d'optimisation fondées sur la manipulation de code de bas niveau peuvent être appliquées. Ce domaine fait l'objet d'une importante activité.

2.5 Note historique

Les préoccupations architecturales dans la conception du logiciel apparaissent vers la fin des années 1960. Le système d'exploitation THE [Dijkstra 1968] est un des premiers exemples de système complexe conçu comme une hiérarchie de machines abstraites. La notion de programmation par objets est introduite dans le langage Simula-67 [Dahl et al. 1970]. La construction modulaire, une approche de la composition systématique de programmes comme un assemblage de parties, apparaît à cette période. Des règles de conception développées pour l'architecture et l'urbanisme [Alexander 1964] sont transposées à la conception de programmes et ont une influence significative sur l'émergence des principes du génie logiciel [Naur and Randell 1969].

La notion de patron de conception vient de la même source, une dizaine d'années plus tard [Alexander et al. 1977]. Avant même que cette notion soit systématiquement utilisée, les patrons élémentaires décrits dans le présent chapitre sont identifiés. Des formes simples d'enveloppes (*wrappers*) sont développées pour convertir des données depuis un format vers un autre, par exemple dans le cadre des systèmes de bases de données, avant d'être utilisées pour transformer les méthodes d'accès. Une utilisation notoire des intercepteurs est la réalisation du premier système réparti de gestion de fichiers, Unix United [Brownbridge et al. 1982] : une couche logicielle interposée à l'interface des appels système Unix permet de rediriger de manière transparente les opérations sur les fichiers distants. Cette méthode sera plus tard étendue [Jones 1993] pour inclure du code utilisateur dans les appels système. Des intercepteurs en pile, côté client et côté serveur, sont introduits dans [Hamilton et al. 1993] sous le nom de *subcontracts*. Diverses formes de mandataires sont utilisés pour réaliser l'exécution à distance, avant que le patron soit identifié [Shapiro 1986]. Les usines semblent être d'abord apparues dans la conception d'interfaces graphiques (par exemple [Weinand et al. 1988]), dans lesquelles un grand nombre d'objets paramétrés (boutons, cadres de fenêtres, menus, etc.) sont créés dynamiquement.

L'exploration systématique des patrons de conception de logiciel commence à la fin des années 1980. Après la publication de [Gamma et al. 1994], l'activité se développe dans ce domaine, avec la création de la série des conférences PLoP [PLoP] et la publication de plusieurs livres spécialisés [Buschmann et al. 1995, Schmidt et al. 2000, Völter et al. 2002].

L'idée de la programmation réflexive est présente sous diverses formes depuis les origines (par exemple dans le mécanisme d'évaluation des langages fonctionnels tels que Lisp). Les premiers essais d'usage systématique de cette notion datent du début des années 1980 (par exemple le mécanisme des métaclasses dans Smalltalk-80) ; les bases du calcul réflexif sont posées dans [Smith 1982]. La notion de protocole à méta-objets [Kiczales et al. 1991] est introduite pour le langage CLOS, une extension objet de Lisp. L'intergiciel réflexif [Kon et al. 2002] fait l'objet de nombreux travaux depuis le milieu des années 1990, et commence à s'introduire dans les systèmes commerciaux (par exemple via le standard CORBA pour les intercepteurs portables).

Bibliographie

[Alexander 1964] Alexander, C. (1964). *Notes on the Synthesis of Form.* Harvard University Press.

[Alexander et al. 1977] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press. 1216 pp.

[ASM 2002] ASM (2002). ASM: a Java Byte-Code Manipulation Framework. The ObjectWeb Consortium, `http://www.objectweb.org/asm/`.

[BCEL] BCEL. Byte Code Engineering Library. http://jakarta.apache.org/bcel.

[Beugnard et al. 1999] Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making Components Contract Aware. *IEEE Computer*, 32(7):38–45.

[Bieber and Carpenter 2002] Bieber, G. and Carpenter, J. (2002). Introduction to Service-Oriented Programming. http://www.openwings.org.

[Brownbridge et al. 1982] Brownbridge, D. R., Marshall, L. F., and Randell, B. (1982). The Newcastle Connection — or UNIXes of the World Unite! *Software – Practice and Experience*, 12(12):1147–1162.

[Bruneton 2001] Bruneton, É. (2001). *Un support d'exécution pour l'adaptation des aspects non-fonctionnels des applications réparties*. PhD thesis, Institut National Polytechnique de Grenoble.

[Buschmann et al. 1995] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1995). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. 467 pp.

[CACM 1993] CACM (1993). *Communications of the ACM*, special issue on concurrent object-oriented programming. 36(9).

[Campbell and Habermann 1974] Campbell, R. H. and Habermann, A. N. (1974). The specification of process synchronization by path expressions. In Gelenbe, E. and Kaiser, C., editors, *Operating Systems, an International Symposium*, volume 16 of *LNCS*, pages 89–102. Springer Verlag.

[Chakrabarti et al. 2002] Chakrabarti, A., de Alfaro, L., Henzinger, T. A., Jurdzinski, M., and Mang, F. Y. (2002). Interface Compatibility Checking for Software Modules. In *Proceedings of the 14th International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 428–441. Springer-Verlag.

[Dahl et al. 1970] Dahl, O.-J., Myhrhaug, B., and Nygaard, K. (1970). The SIMULA 67 common base language. Technical Report S-22, Norwegian Computing Center, Oslo, Norway.

[Dijkstra 1968] Dijkstra, E. W. (1968). The Structure of the THE Multiprogramming System. *Communications of the ACM*, 11(5):341–346.

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 416 pp.

[Hagimont and De Palma 2002] Hagimont, D. and De Palma, N. (2002). Removing Indirection Objects for Non-functional Properties. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*.

[Hamilton et al. 1993] Hamilton, G., Powell, M. L., and Mitchell, J. G. (1993). Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, volume 27 of *Operating Systems Review*, pages 69–79, Asheville, NC (USA).

[Hoare 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585.

[Jones 1993] Jones, M. B. (1993). Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC (USA).

[Kiczales 1996] Kiczales, G. (1996). Aspect-Oriented Programming. *ACM Computing Surveys*, 28(4):154.

[Kiczales et al. 1991] Kiczales, G., des Rivières, J., and Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*. MIT Press. 345 pp.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proceedings of ECOOP 2001*, volume 2072 of *LNCS*, pages 327–355, Budapest, Hungary. Springer-Verlag.

[Kon et al. 2002] Kon, F., Costa, F., Blair, G., and Campbell, R. (2002). The case for reflective middleware. *Communications of the ACM*, 45(6):33–38.

[Kramer 1998] Kramer, R. (1998). iContract - The Java Design by Contract Tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS) Conference*, pages 295–307.

[Lindholm and Yellin 1996] Lindholm, T. and Yellin, F. (1996). *The Java Virtual Machine Specification.* Addison-Wesley. 475 pp.

[Meyer 1992] Meyer, B. (1992). Applying Design by Contract. *IEEE Computer*, 25(10):40–52.

[Naur and Randell 1969] Naur, P. and Randell, B., editors (1969). *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee, 7-11 Oct. 1968.* Scientific Affairs Division, NATO. 231 pp.

[Pawlak et al. 2001] Pawlak, R., Duchien, L., Florin, G., and Seinturier, L. (2001). JAC : a flexible solution for aspect oriented programming in Java. In Yonezawa, A. and Matsuoka, S., editors, *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 1–24, Kyoto, Japan. Springer-Verlag.

[PLoP] PLoP. The Pattern Languages of Programs (PLoP) Conference Series. http://www.hillside.net/conferences/plop.htm.

[RM 2000] RM (2000). *Workshop on Reflective Middleware.* Held in conjunction with Middleware 2000, 7-8 April 2000. http://www.comp.lancs.ac.uk/computing/RM2000/.

[Schmidt et al. 2000] Schmidt, D. C., Stal, M., Rohnert, H., and Buschmann, F. (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* John Wiley & Sons. 666 pp.

[Shapiro 1986] Shapiro, M. (1986). Structure and encapsulation in distributed systems: The proxy principle. In *Proc. of the 6th International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA). IEEE.

[Smith 1982] Smith, B. C. (1982). *Reflection And Semantics In A Procedural Language.* PhD thesis, Massachusetts Institute of Technology. MIT/LCS/TR-272.

[Tatsubori et al. 2001] Tatsubori, M., Sasaki, T., Chiba, S., and Itano, K. (2001). A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *LNCS*, pages 236–255. Springer Verlag.

[Völter et al. 2002] Völter, M., Schmid, A., and Wolff, E. (2002). *Server Component Patterns.* John Wiley & Sons. 462 pp.

[Weinand et al. 1988] Weinand, A., Gamma, E., and Marty, R. (1988). ET++ - An Object-Oriented Application Framework in C++. In *Proceedings of OOPSLA 1988*, pages 46–57.