Teoria współbieżności

Laboratorium 7

Wzorce projektowe dla programowania współbieżnego

Wyzwania dla projektantów sieciowych i współbieżnych aplikacji

1. Schmidt et al. wymieniają cztery istotne aspekty sieciowych i współbieżnych aplikacji:
    - Dostęp do usługi i konfiguracja
    - Obsługa zdarzeń
    - Współbieżność
    - Synchronizacja
2. **Dostęp do usługi i konfiguracja** (Service access and configuration): związany z tym, że komponenty sieciowej aplikacji współpracują przy pomocy różnych mechanizmów i protokołów komunikacji (np. IPC, TCP/IP, TELNET/FTP/HTTP, RPC). (Poza naszym zainteresowaniem.)
3. **Obsługa zdarzeń** (Event handling): związana z aplikacjami sterowanymi zdarzeniami (event-driven). Aplikacje takie wyróżniają się od aplikacji 'tradycyjnych' trzema charakterystycznymi cechami:
    - Przetwarzanie w aplikacji jest wywoływane zdarzeniami, które występują asynchronicznie. Zdarzenia te mogą pochodzić od sterowników urządzeń, portów I/O, sensorów, klawiatury i myszy, sygnałów, etc.
    - Większość zdarzeń powinna być obsłużona szybko (np. ze względu na czas odpowiedzi).
    - Aplikacje sterowane zdarzeniami nie mają wpływu na kolejność, w której przychodzą zdarzenia. Dlatego może być konieczna kontrola i wykrywanie nielegalnych transakcji.
4. **Współbieżność** (concurrency) dotyczy mozliwosci wykonywania wielu zadań jednocześnie przez wiele procesów/ wątków, np. jednoczesnej obsługi żądań wielu klientów przez serwer. Użycie wątków może poprawić responsywność, efektywność i strukturę (design) aplikacji.
5. **Synchronizacja** (synchronization): programowanie

współbieżne jest trudniejsze niż sekwencyjne ze względu na konieczność synchronizacji dostępu do zasobów dzielonych.

Wzorce współbieżności (Concurrency patterns)

1. Aktywny obiekt (Active object)
2. Monitor object
3. Half-Sync / Half-Async
4. Leader-Follower
5. Thread-Specific Storage

Wzorzec Active Object

Aktywny obiekt *oddziela wykonanie metody od wywołania metody* aby poprawić współbieżność i uprościć synchronizowany dostęp do obiektów, które są umieszczone w swoich własnych wątkach.

Alternatywna nazwa

Concurrent object

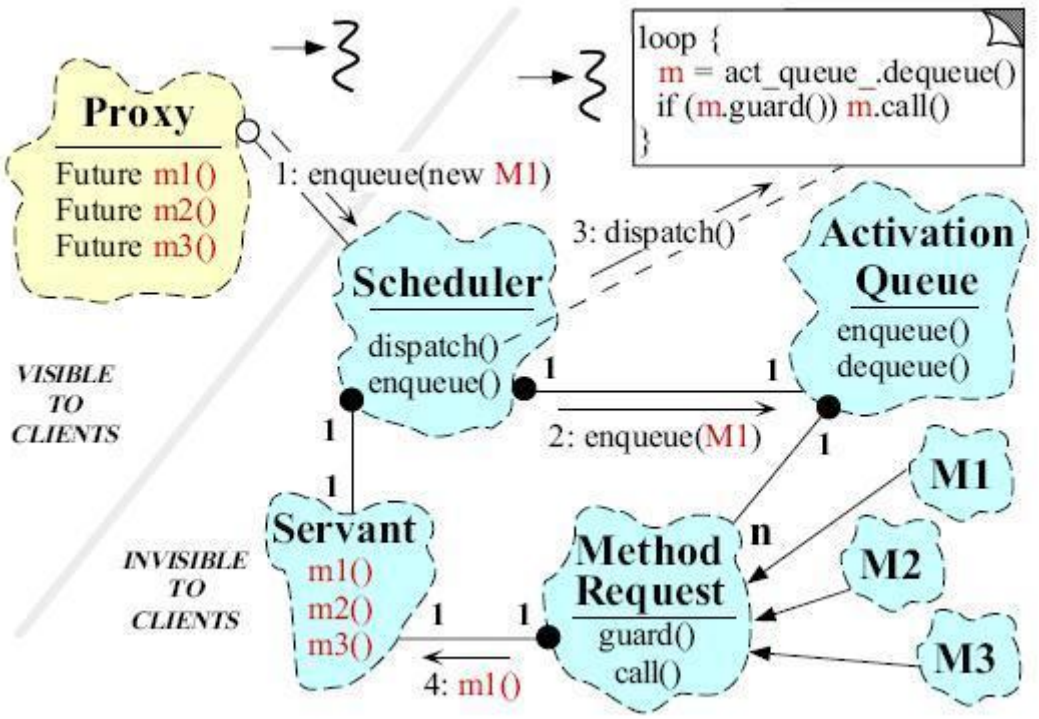
Kontekst

Klienci, którzy odwołują się do obiektów wykonujących się w osobnych wątkach.

Rozwiązanie

Oddzielenie wywołania metody od jej wykonania. Wywołanie metody odbywa się w wątku klienta, lecz jej wykonanie w osobnym wątku - pracownika. Powinno to być tak zaprojektowane, żeby z punktu widzenia klienta wyglądało to na zwykłe wywołanie metody.

Struktura



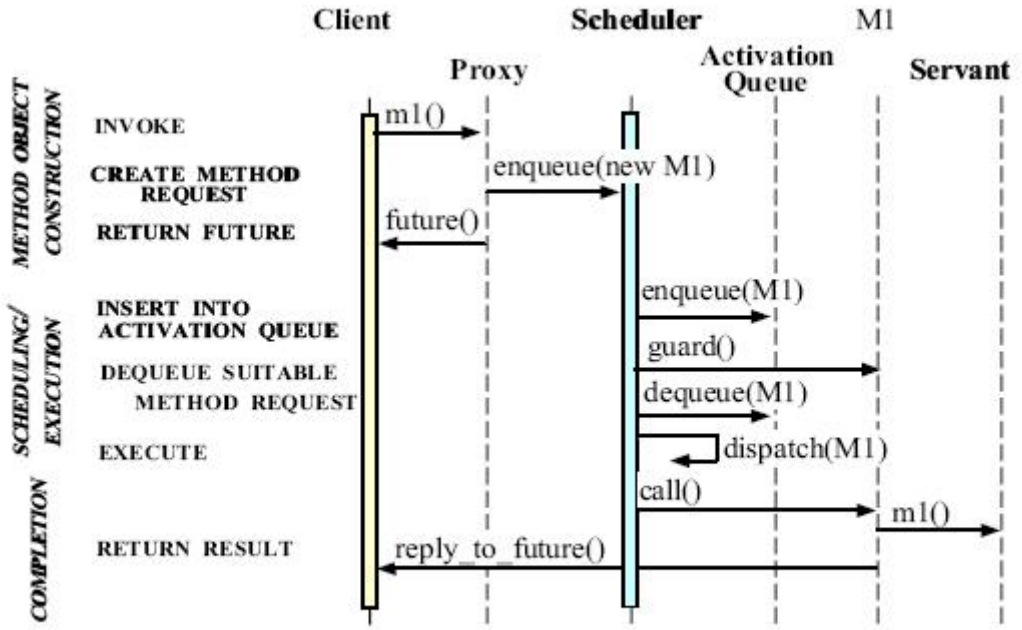
Active object składa się z sześciu składników:

1. **Proxy** (Pośrednik). Reprezentuje interfejs obiektu. Udostępnia interfejs dla klientów, przy pomocy którego wywołują oni publicznie dostępne metody aktywnego obiektu. Wywołanie takiej metody powoduje utworzenie obiektu *Method Request* i umieszczenie go w kolejce *Activation Queue* należącej do *Schedulera*.
2. **Servant** (Pracownik). Dostarcza implementacji obiektu. Implementuje metody zdefiniowane w *Proxy* i odpowiednim *Method Request*. Metoda *Servanta* jest wywołana gdy odpowiadające jej żądanie metody (*Method Request*) jest wykonane przez *Scheduler*. *Servant* wykonuje się w wątku *Schedulera*. *Servant* może dostarczać dodatkowych metod, które mogą posłużyć do implementacji strażników w *Method Request*
3. **Method Request** (Żądanie metody). Używane jest do przekazanie *kontekstu wywołania* metody (np. parametrów) z Proxy do Schedulera uruchomionego w osobnym wątku. Abstrakcyjna klasa *Method Request* definiuje interfejs dla wykonywania metod Aktywnego Obiektu. Interfejs ten zawiera również metody *strażników* (guard), które są mogą być użyte do sprawdzenia, czy warunki związane z synchronizacją są spełnione.
4. **Activation Queue** (Kolejka aktywacji). Zawiera bufor oczekujących *Method Request* utworzonych przez *Proxy*. Jest zasobem dzielonym dla wątków klienta i pracownika - pierwszy

jest producentem żądań metody, drugi ich konsumentem (przez Scheduler).

5. **Scheduler** (Zarządca). Wykonuje się w osobnym wątku, zarządzając *Activation Queue*. Decyduje, które *Method Request* ma być zdjęte z kolejki i wykonane w *Servant* implementującym metodę. Ta decyzja oparta jest o różne kryteria, np. kolejność umieszczenia metod w kolejce, spełnienie pewnych warunków związanych z synchronizacją (np. zwolnienie miejsca w buforze). Uwarunkowania związane z synchronizacją są zwykle sprawdzane przy użyciu metod - *strażników*.
6. **Future** (Zmienna terminowa). Pozwala klientowi pobranie wyniku wykonania metody, gdy *Servant* zakończy jej wykonanie. Po wywołaniu metody w *Proxy*, obiekt Future jest od razu zwracany. Future rezerwuje miejsce na wyniki wywołania metody. Gdy klient chce te wyniki pobrać, albo się blokuje do czasu pojawienia się wyników, albo periodycznie sprawdza przez *polling*.

Przebieg wywołania



Zadania:

Zaimplementować bufor jako aktywny obiekt (Producenci-Konsumenci)

Wskazówki:

1. Pracownik powinien implementować samą kolejkę (bufor) oraz

dodatkowe metody (czyPusty etc.), które pomogą w implementacji strażników. W klasie tej powinna być tylko funkcjonalność, ale nie logika związana z synchronizacją.
2. Dla każdej metody aktywnego obiektu powinna być specjalizacja klasy MethodRequest. W tej klasie m.in. zaimplementowana jest metoda guard(), która oblicza spełnienie warunków synchronizacji (korzystając z metod dostarczonych przez Pracownika).
3. Proxy wykonuje się w wątku klienta, który wywołuje metodę. Tworzenie Method request i kolejkowanie jej w Activation queue odbywa się również w wątku klienta. Servant i Scheduler wykonują się w osobnym (oba w tym samym) wątku.

Bibliografia

1. R.G. Lavender, D.C. Schmidt: Active Object