ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1995

Seria: INFORMATYKA z.28 Nr kol. 1270

Leszek BORZEMSKI
Mariusz FRAŚ
Dariusz GAJEWSKI
Ziemowit NOWAK
Mirosław SZCZYPIŃSKI

BUDOWA I EKSPLOATACJA APLIKACJI W SIECIOWYM ŚRODOWISKU X WINDOWS*)

Streszczenie. W artykule przedstawiono ogólne zasady programowania w sieciowym środowisku graficznym X Windows. Omówiono architekturę systemu, podstawowe zasady używania bibliotek Xlib, XToolkit. Przedstawione zostały również pewne aspekty poprawnej konfiguracji środowiska użytkownika systemu X Windows. Zaprezentowano moliwości budowy aplikacji z wykorzystaniem pakietu DevGuide.

BUILDING AND EKSPLOATATION APLICATIONS IN X WINDOWS NETWORK ENVIRONMNET

Summary. The purpose of this paper is to present general ideas of programming in a network graphical environment called X Windows. In particular the architecture and programming environment were described. We present basic libraries Xlib, XToolkit and a short overview how to use them. In paper there are also described some aspects of proper environment configuration. The DevGuide package for quick developement of X Windows-based aplications is also presented.

*) Pracę wykonano w ramach grantu KBN nr 350006.

DIE STRUKTUR UND AUSWERTUNG VON ANWENDUNGEN IN XWINDOWS NETZWERK-UMGEBUNG

Zusammenfassung. Die allgemeine Prinzipien von Programmierung in graphische X Windows Netzwerk-Umgebung sind dargestellt. Unter beachtung wurde die System-Architekturen und daraus entstehende Konsekwenzen für Programmierer diskutiert. Auch die Unterschiede von Regeln und Handbibliothek von Programmen zu verwenden ist behandelt, wie z.B. Xlib, XToolkit. Die Überlegungen von gewisse Aspekte der korrekte Konfigurationen und Software-gestütute Programmierung von DevGuide in X Windows-Umgebung diskutiert wurde.

1. Wstęp

System X Windows powstał w MIT w roku 1984 w trakcie prac nad projektem Athena, którego celem było stworzenie standardu systemu graficznego do integracji systemów i aplikacji pochodzących od różnych producentów. Projektem i implementacją pierwszych 10 wersji systemu X zajmowało się głównie trzy osoby: Robert Scheifler (MIT Laboratory for Computer Science), Jim Gettys (DEC), Ron Newman (MIT). Wersja 11 jest już dziełem wielu osób i organizacji. W 1988 roku powołano do życia X konsorcjum m.in. z udziałem AT&T, HP, Sun zaimujące się rozwojem systemu.

W systemie X, jak w wielu innych systemach graficznych, każda z aplikacji może tworzyć na ekranie monitora okna, które można przesuwać, zamykać, zmieniać ich rozmiary. Podstawowym urządzeniem wejściowym jest mysz umożliwiająca sterowanie wykonywaniem aplikacji, często bez udziału klawiatury. Okna w systemie X mają strukturę hierarchiczną. Okno-potomek może mieć większe rozmiary niż okno-przodek. Okno otoczone jest ramką. Ramka może być zerowej grubości. Z każdym z okien związany jest układ współrzędnych.

Przenośność systemu X uzyskano definiując go jako protokół, w którym opracowano system komunikatów poleceń i odpowiedzi w miejsce tradycyjnych wywołań funkcji. Przesyłane komunikaty tworzą tzw. protokół X. W systemie X kontrolę nad ekranem, klawiaturą i myszą sprawuje, tzw. X serwer, zwany X usługodawcą. Aplikacja wykonuje odpowiednie działania na ekranie monitora wysyłając odpowiednie polecenia X protokołu do serwera systemu X. X serwer może kontrolować wiele ekranów monitorów jednocześnie, także zdalnych (poprzez sieć komputerową).

W celu zrozumienia sposobu programowania w systemie X Windows konieczne jest zrozumienie "callback-owego" stylu programowania, które różni się istotnie od stylu tradycyjnego. X Windows jest systemem opartym na przesyłaniu wiadomości (tzw. notification-based system). Notyfikator (ang. notifier), bo tak nazywa się moduł kontrolny, działa jako proces użytkownika. Jego zadaniem jest rejestracja oraz formatowanie zdarzeń a następnie rozprowadzanie ich do innych części systemu. W konwencjonalnym, interakcyjnym stylu programowania, główna pętla kontrolująca przebieg wykonywania programu znajduje się w aplikacji. Np. edytor czyta znak i podejmuje pewną akcję w zależności od znaczenia, jakie mu zostało przypisane w programie. Jeżeli jest to znak końca, to program kończy pętlę oraz swoje działanie, jeśli nie, to czyta kolejny znak. System oparty na mechanizmie "Notifier-a"

zmienia tę "prostoliniową" strukturę. Główna pętla sterująca przebiegiem programu nie jest umieszczona w aplikacji, lecz mieści się w programie wykonywanym przez notyfikator. To notyfikator czyta dane z wejścia, zmienia je na odpowiedni format zdarzenia i powiadamia odpowiednie procedury, które zostały zarejestrowane w procesie Notifier-a. Procedury te nazywane są "notify procs" lub "callback procs". Z punktu widzenia programisty istotne jest pojęcie X-zdarzenia. Jest to struktura danych generowana przez usługodawcę, informująca o tym, co istotnego z punktu widzenia aplikacji zdarzyło się w systemie (ruch myszy, dotknięcie klawiatury, zmiana położenia okna itp.). Według tego schematu X-aplikacja podzielona jest na dwie części: deklarującą obiekty i zdarzenia i kontrolującą przepływ informacji. Notyfikator oprócz sygnałów od aplikacji i usługodawcy może przekazywać również informacje od systemu opereracyjnego i innych procesów-klientów. Biblioteką pozwalającą na bezpośrednie generowanie i czytanie poleceń protokołu X jest Xlib. Jest to podstawowa, a zarazem najistotniejsza część środowiska programowego. Za pomocą tej biblioteki możliwe jest tworzenie kompletnych aplikacji. Należy podkreślić, że biblioteka nie narzuca żadnych sztywnych wymagań konstrukcji aplikacji (np. wyglądu). Jest to najniższa warstwa oprogramowania.

W środowisku X Windows stworzono wiele bibliotek narzędziowych pozwalających programiście na korzystanie z usług wyższego poziomu (np. XToolkit). Pozwalają one na korzystanie z gotowych procedur, uwalniając od żmudnego kodowania poszczególnych elementów aplikacji. W środowisku X Windows dostępne są także narzędzia, pozwalające na automatyczne generowanie interfejsu graficznego, bądź wspomagające jego tworzenie. Opracowano szereg narzędzi programistycznych wyższego poziomu. Do nich należy system Devguide. Devguide jest narzędziem programisty wspomagającym tworzenie interfejsu graficznego aplikacji z użytkownikiem w systemie SunOs. Umożliwia budowę interfejsu, jego modyfikację, testowanie i generację kodu źródłowego w języku C.

2. Środowisko pracy aplikacji - protokół systemu X Windows

X protokół może zostać zbudowany na dowolnym strumieniu umożliwiającym przepływ strumienia bajtów między procesami, także na strumieniu danych przesyłanych poprzez sieć komputerową. W celu zapewnienia poprawnej interpretacji przesyłanego strumienia bajtów przed właściwym nawiązaniem połączenia pomiędzy aplikacją (klientem) a serwerem (usługodawcą) wysyłany jest bajt identyfikujący kolejność przesyłania bajtów. Musi to być albo liczba 102, albo 154. 102 oznacza, że w strumieniu danych jako pierwszy jest przesyłany najbardziej znaczący bajt, 154 - jeśli jest to bajt najmniej znaczący.

Informacje przesyłane są w jednym z czterech formatów:

1. **Format polecenia** (ang. request format)

Każde polecenie identyfikowane jest przez 8-bitowy kod operacji (ang. opcode) umieszczany w 4 B nagłówku ramki polecenia. Oprócz tego w ramce umieszczone jest pole długości całej ramki i oczywiście przekazywane dane (zero lub więcej bajtów).

2. **Format odpowiedzi** (ang. reply format)

W skład ramki odpowiedzi wchodzą co najmniej 32 B i dodatkowo zero lub więcej bajtów danych. Każda odpowiedź zawiera także 16-bitowy numer sekwencji ramki polecenia.

3. **Format błędu** (ang. error format)

Błędy przesyłane są w 32 B ramkach. Każda ramka zawiera 8 b kod błędu, a także numery polecenia (ang. minor, major opcodes) oraz najmniej znaczące 16 b numeru sekwencji odpowiedniego polecenia.

4. **Format zdarzenia** (ang. events format)

Zdarzenia przesyłane są w 32 B ramkach. Każda zawiera 8 b pole typu kodu operacji, najmniej znaczące 16 b numeru sekwencji ostatnio wykonanego lub wykonywanego przez serwer polecenia. Najstarszy bit w ramce jest ustawiany, jeżeli zdarzenie zostało wygenerowane na skutek przesłania polecenia SendEvent.

W trakcie nawiązywania połączenia z serwerem klient przesyła do serwera następujące informacje:
- numer protokołu X (ang. protocol-major-version, protocol-minor-version),
- nazwa protokołu autoryzacji (ang. authorization-protocol-name),
- dane dla autoryzacji (ang. authorization-protocol-data).

Numery wersji protokołu odnoszą się do numerów wersji, których klient oczekuje od serwera, a pole danych zawiera dane potrzebne do dokonania autoryzacji. Klient w odpowiedzi otrzymuje:
- kod powrotu (ang. succes),
- numery protokołu (ang. protocol-major-version, protocol-minor-version),
- długość (ang. length).

Numery protokołu odnoszą się do wersji protokołu X serwera, może on być różny od numeru wersji protokołu klienta. Serwer może odmówić nawiązania połączenia, jeśli dana wersja protokołu klienta nie jest akceptowalna przez serwer. Pole długości (ang. length) określa liczbę elementów 4-bajtowych, które są przesyłane w odpowiedzi. Jeśli autoryzacja zostanie dokonana, to przesyłane są dodatkowe dane, np. nazwa producenta implementacji serwera, opisujące dokładnie parametry serwera protokołu X.

Przykład. Polecenia i zdarzenia protokołu X

Polecenia:
CreateWindow - tworzy niezamapowane okno i przypisuje mu identyfikator.
QueryFont - zwraca informację o fontach.
Bell - sygnał dzwonka.

Zdarzenia:
KeyPress - naciśnięcie klawisza.

Zamknięcie połączenia pomiędzy klientem a serwerem powoduje m.in. usunięcie zdarzeń związanych z procesem-klientem, usunięcie wszystkich okien klienta, zwolnienie zasobów itp.

W trakcie przesyłania informacji pomiędzy klientem a serwerem serwer nie musi buforować więcej niż jedno polecenie na połączenie w jednej chwili. Polecenia są realizowane w kolejności napływu. Na wykonanie polecenia składa się walidacja argumentów, zbieranie danych do przesłania odpowiedzi, generacja i kolejkowanie zdarzeń. W skład wykonania polecenia nie wchodzi natomiast czas transmisji odpowiedzi i zdarzeń. Proces klienta jest informowany o wszystkich zaistniałych wyjątkach, które napłynęły do czasu wykonania polecenia lub zaistniały w czasie jego wykonywania przed przesłaniem odpowiedzi lub kodu błędu. Operacje przez serwer wykonywane są wg reguły "wszystko albo nic", tj. polecenia albo całkowicie zostaną wykonane prawidłowo, albo nie wykonane zostanie żadne działanie

Nie jest więc możliwe częściowe wykonanie polecenia i wyświetlenie na ekranie nie dokończonego efektu działania jakiegoś polecenia.

3. Biblioteka najniższego poziomu budowy aplikacji - Xlib

Biblioteka Xlib stanowi interfejs niskiego poziomu w postaci biblioteki funkcji do protokołu systemu X Windows. Większość tych funkcji wykonuje swoje operacje na buforze roboczym, a wygląd ekranu jest aktualizowany asynchronicznie przez X serwer. Działanie synchroniczne można uzyskać przez wywołanie funkcji XSync, która wymusza na serwerze przetworzenie wszystkich poprzednio zanotowanych zdarzeń i aktualizację zawartości ekranu. Aktualizacja zawartości ekranu na podstawie zawartości bufora następuje także przy każdym wywołaniu funkcji, która zwraca wartość jakiejś danej od X serwera lub wywołanie funkcji powoduje oczekiwanie na wprowadzenie danych. System X nie gwarantuje ochrony zawartości okien. Proces-klient musi być przygotowany do obsługi zdarzenia Expose generowanego w przypadku konieczności odświeżenia zawartości okna. Zdarzenie to informuje proces o konieczności odświeżenia swojego okna.

Wiele funkcji biblioteki Xlib zwraca wartość typu całkowitego będącą deskryptorem zasobu X serwera, umożliwiając w ten sposób dostęp do zasobów X serwera. Przykładowe typy zasobów to: Window, Font, Pixmap, Colormap, Cursor, GContext. Typy te zdefiniowane są w pliku <X11/X.h>. Zasoby te tworzone są przez wywołania odpowiednich funkcji, a usuwane są przez wywołania innych funkcji lub przez zamknięcie połączenia z serwerem. Większość z tych zasobów może być współdzielona przez różne aplikacje. Fonty i kursory (ang. cursors) są współdzielone automatycznie. Fonty są ładowane i odładowywane dynamicznie w miarę potrzeb i są współdzielone przez wiele klientów serwera. Fonty są czasem przechowywane w pamięci podręcznej. Xlib nie pozwala na współdzielenie kontekstu graficznego pomiędzy aplikacjami.

Procesy klientów informowane są o zdarzeniach. Zdarzenia mogą być generowane synchronicznie lub asynchronicznie (np. z klawiatury). Proces klienta otrzymuje informację o zdarzeniach na własne żądanie i musi być przygotowany do ich obsługi bądź ich ignorowania. Zdarzenia wejściowe (ang. input events), powstałe, np. po naciśnięciu klawisza lub ruchu myszki, docierają asynchronicznie do serwera i są kolejkowane do chwili jawnego wywołania funkcji, np. XNextEvent lub XWindowEvent.

Niektóre funkcje zwracają informacje o statusie wykonanej operacji. W przypadku wystąpienia błędu funkcja zwraca wartość zero. X serwer zgłasza błędy X protokołu w chwili ich wystąpienia. Ponieważ Xlib nie komunikuje się bezpośrednio z serwerem, lecz za pomocą bufora, błędy mogą być sygnalizowane znacznie później niż wystąpiły. Dla celów testowych Xlib ma wbudowane mechanizmy do synchronicznego raportowania wystąpienia błędu. Jeśli na poziomie Xlib zostanie wykryty błąd, to wywołana zostanie funkcja obsługi błędu napisana przez programistę, a jeśli takiej funkcji nie ma, komunikat o błędzie zostanie wydrukowany na monitorze, a realizacja programu jest przerywana.

3.1. Połączenie z serwerem

Przed wykonaniem jakiejkolwiek operacji na ekranie monitora proces musi nawiązać połączenie z serwerem systemu X zarządzającym ekranem monitora. Do nawiązania łączności z serwerem można użyć funkcji XOpenDisplay:

```
Display *XOpenDisplay(display_name)
    char *display_name;
```

Ciąg znaków zawarty w zmiennej display_name określa pełną nazwę ekranu monitora. Gdy wartość jej jest równa NULL, odczytywana jest wartość zmiennej systemowej DISPLAY. Zmienna ta ma następujący format:

```
nazwa_hosta:numer.numer_ekranu
```

np. ua:0.0.

Parametry te oznaczają odpowiednio:
nazwa_hosta - nazwę komputera, do którego jest podłączony monitor,
numer - jest numerem serwera systemu X, który kontroluje ten monitor,
numer_ekranu - jest numerem monitora (do jednego komputera może być ich przyłączonych wiele).

Funkcja XOpenDisplay zwraca wskaźnik na strukturę typu Display zawierającą wszystkie podstawowe informacje o serwerze. Funkcja ta nawiązuje połączenie lokalne lub sieciowe (TCP/IP, DECnet) z X serwerem. Struktura Display jest zdefiniowana w pliku <X11.Xlib.h>. W przypadku niepowodzenia funkcja zwraca wartość NULL. Do odczytu danych z tej struktury w Xlib zdefiniowanych zostało wiele funkcji i makr. Aplikacja nigdy nie powinna bezpośrednio modyfikować pól struktur Display i Screen.

Aplikacja powinna przekazywać dane X serwerowi w odpowiednim porządku i formacie. Xlib zawiera szereg makr upraszczających to zadanie, np. XImageByteOrder.

Do celów testowych w Xlib umieszczona została funkcja XNoOp odpowiadająca poleceniu protokołu NoOperation:

```
XNoOp(display)
    display *display;
```

W szczególności, funkcji tej można użyć do testów poprawności wymiany danych pomiędzy serwerem systemu a aplikacją.

W trakcie pracy aplikacja ma możliwość zwolnienia pamięci używanej przez Xlib przez wywołanie funkcji XFree:

```
XFree(data)
    char *data;
```

gdzie data jest wskaźnikiem na zwalniany obszar danych.

Zamknięcie połączenia z serwerem następuje po wykonaniu funkcji XCloseDisplay:

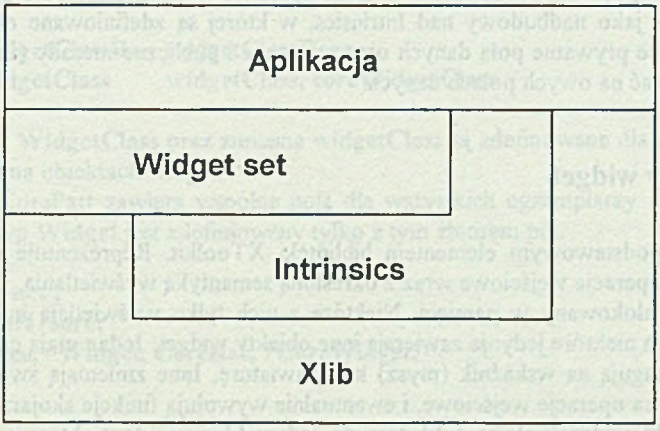
```
XCloseDisplay(display)
    Display *display;
```

Funkcja ta zamyka połączenie, usuwa wszystkie okna aplikacji oraz deskryptory zasobów (Window, Font, Pixmap, Colormap, Cursor, GContext).

4. Wykorzystanie biblioteki XToolkit do budowania aplikacji

XToolkit tworzy biblioteka **Intrinsics** oraz zbiór tzw. **widget'ów (obiektów widget)**. Intrinsics dostarcza podstawowych mechanizmów umożliwiających budowę zbiorów widget oraz środowiska aplikacji. Jest to narzędzie wyższego poziomu napisane w stylu zorientowanym obiektowo, używane do budowy komponentów interfejsu użytkownika. Widget'y to podstawowe elementy służące do realizacji okien. Są to obiekty realizujące elementy interfejsu użytkownika (np. Scrollbar widget). Dostępnych jest kilka zbiorów obiektów typu widget, które reprezentują pewne konwencje interfejsu użytkownika (np. Athena Widgets Set).

Biblioteka Intrinsics jest warstwą nałożoną na bibliotekę Xlib i dostarcza dodatkowych, prostych mechanizmów (funkcje i struktury) pomocnych przy tworzeniu aplikacji i wyższych warstw w strukturze bibliotek funkcji systemu X. Typowa aplikacja, napisana z wykorzystaniem XToolkit, w małym stopniu korzysta bezpośrednio z biblioteki Xlib. Jest ona kompozycją zbioru obiektów widget, podzbioru funkcji Intrinsics oraz funkcji Xlib.



Rys.1. Architektura warstwowa bibliotek funkcji systemu X
Fig.1. Tiered architecture of libraries in X window system

XToolkit dostarcza programistom narzędzi do budowy interfejsu aplikacji (zbiór obiektów widget) oraz mechanizmów Intrinsics do tworzenia widgetów. Niektóre obiekty widget zbudowane z użyciem Intrinsics mogą być wspólne dla wielu aplikacji, inne mogą być specyficzne dla jednej aplikacji. Ponadto Intrinsics pomaga w pośrednim zarządzaniu:

1. inicjalizacją XToolkit'a,
2. obiektami widget,
3. pamięcią,
4. zdarzeniami,
5. geometrią obiektów,
6. wyglądem zewnętrznym aplikacji,
7. wybieraniem opcji,

8. zasobami i konwersją zasobów,
9. translacją zdarzeń,
10. kontekstem graficznym,
11. kolorami (mapami pixelowymi - pixmaps),
12. błędami i ostrzeżeniami.

Intrinsics ma architekturę wystarczająco elastyczną do obsługi różnorodnych warstw interfejsów aplikacji. Przy tym Intrinsics zapewnia pełną funkcjonalność i swobodę programowania, stylistycznie i funkcjonalnie pełną zgodność z podstawami systemu X Windows oraz przenośność pod względem jezyka, architektury komputera i systemu operacyjnego.

Aplikacje korzystające z mechanizmów tej biblioteki wymagają włączenia następujących zbiorów nagłówkowych: X11.Intrinsic.h, X11.StringDefs.h. Dodatkowo mogą być również włączone: X11.Atom.h, X11.Shell.h. Jednak finalnie wystarczy włączyć X11.IntrinsicP.h, który zawiera wszystkie wyżej wymienione zbiory nagłówkowe. Oprócz włączenia powyższego zbioru nagłówkowego każda aplikacja korzystająca z obiektów klasy widget musi mieć deklarację włączającą dany obiekt (np. #include <Xm / Scrollbar.h >).

Widać więc, że aplikacja napisana dla środowiska X Windows może być oparta bezpośrednio na bibliotece Xlib, jak również być napisana z wykorzystaniem narzędzi wyższego poziomu, np. tzw. XToolkit'a. Zwykle aplikacja wykorzystuje obie biblioteki, tzn. Xlib i XToolkit - jako nadbudowy nad Intrinsics, w której są zdefiniowane obiekty zwane widget, zawierające prywatne pola danych oraz prywatne i publiczne metody (funkcje), które pozwalają operować na owych polach danych.

4.1. Obiekty widget

Widget jest podstawowym elementem biblioteki XToolkit. Reprezentuje on X-okno i powiązane z nim operacje wejściowe wraz z określoną semantyką wyświetlania. Obiekt widget jest dynamicznie alokowany w pamięci. Niektóre z nich tylko wyświetlają informację (np. tekst lub grafikę), a niektóre jedynie zawierają inne obiekty widget. Jedne mają charakter tylko wyjściowy, nie reagują na wskaźnik (mysz) lub klawiaturę. Inne zmieniają swoją zawartość właśnie w reakcji na operacje wejściowe, i ewentualnie wywołują funkcje skojarzoną z danym działaniem. Obiekt widget należy dokładnie do jednej **klasy widget**, która jest statycznie alokowana, inicjowana i w której określone są dozwolone operacje. Klasa widget składa się z pól danych i metod. Dane i metody mogą być dziedziczone przez podklasy. Fizycznie, tj. w programie, klasa widget jest reprezentowana przez wskaźnik na odpowiednią strukturę. Zawartość tej struktury jest stała dla wszystkich obiektów danej klasy. Większość z parametrów obiektu widget (np. czcionki, kolory, rozmiary, grubości ramek itd.) może określić programista. **Egzemplarz (instancja)** obiektu widget składa się z dwóch części:
- struktury danych, która zawiera specyficzne dla obiektu wartości (wartości początkowe, które mogą być zmieniane przez programistę poprzez wywołanie odpowiedniej funkcji),
- struktury danych, która zawiera elementy (dane i metody) wspólne (dostępne) dla wszystkich obiektów tej klasy.

Intrinsics dostarcza wiele mechanizmów do budowy i zarządzania obiektami widget. Jednakże całość oparta jest na trzech podstawowych obiektach widget, które są podstawą do budowy własnych obiektów, a tym samym budowy własnego interfejsu dla aplikacji. Te trzy podstawowe klasy to:

- klasa "core widget" (ang. core widget class),
- klasa "composite widget" (ang. composite widget class),
- klasa "constraint widget" (ang. constraint widget class).

4.2. Obiekty "Core widget"

Obiekt "core widget" zawiera definicje pól wspólne dla wszystkich obiektów widget. Wszystkie obiekty widget są podklasami obiektu Core, który jest zdefiniowany przez dwie struktury **CoreClassPart** i **CorePart**. Struktura CoreClassPart zawiera pola wspólne dla wszystkich klas widget. Prototypowa klasa widget **WidgetClass** zawiera tylko te pola. Oto kilka typów zdefiniowanych za pomocą CoreClassPart:

```
typedef struct {
    CoreClassPart    core_class;
} WidgetClassRec, *WidgetClass, CoreClassRec, *CoreWidgetClass;
```

Predefiniowany rekord klasy i wskaźnik dla **WidgetClassRec** to:

```
extern WidgetClassRec  widgetClassRec;
extern WidgetClass     widgetClass, coreWidgetClass
```

Typy **Widget** i **WidgetClass** oraz zmienna **widgetClass** są zdefinowane dla przeprowadzenia ogólnych akcji na obiektach widget.

Struktura CorePart zawiera wspólne pola dla wszystkich egzemplarzy obiektów widget. Prototypowy typ **Widget** jest zdefiniowany tylko z tym zbiorem pól.

```
typedef struct {
    CorePart  core;
} WidgetRec, *Widget, CoreRec, *CoreWidget;
```

4.3. Obiekty "Composite widget"

Obiekty "Composite widget" są podklasą "Core widget" i są przeznaczone jako zasobniki dla innych obiektów widget (tzn. są rodzicami dla innych podklas widget).Obiekty "Composite widget" definiują struktury **CompositeClassPart** i **CompositePart**. W klasie "Composite widget" pola struktury CompositeClassPart występują bezpośrednio po polach klasy Core:

```
typedef struct {
    CoreClassPart       core_class;
    CompositeClassPart composite_class;
} CompositeClassRec, *CompositeWidgetClass;
```

Obiekt "Composite widget" zdefiniowany jest następująco:

```
typedef struct {
    CorePart        core;
    CompositePart   Composite;
} CompositeRec,     *CompositeWidget;
```

4.4. Obiekty "Constraint widget"

"Constraint widget" są podklasą "Composite widget", zawierajacą dodatkowe dane o stanie dla każdego potomka (np. zdefiniowane przez klienta dane o geometrii obiektów potomnych). Obiekt Constraint jest zdefiniowany przez struktury **ConstraintClassPart** i **ConstraintPart**. Klasa "Constraint widget" przedstawia się następująco:

```
typedef struct {
    CoreClassPart       core_class;
    CompositeClassPart  composite_class;
    ConstraintClassPart constraint_class;
} CoustraintClassRec, *CoustraintWidgetClass;
```

a obiekt "Constraint widget":

```
typedef struct {
    CorePart        core
    CompositePart   composite
    ConstrainPart   constrain
} ConstrainRec, *ConstraintWidget;
```

4.5. Drzewo obiektów widget

Kolekcja obiektów widget ma strukturę drzewiastą, której korzeniem jest obiekt widget **shell** (ang. shell widget) zwracany przez funkcję **XtAppCreateShell.** Obiekty widget z jednym lub wieloma potomkami są pośrednimi węzłami tego drzewa, zaś obiekt widget bez potomka liściem drzewa. Obiekty widget mogą być obiektami prostymi (typu primitive) lub złożonymi (typu composite). Oba typy mogą mieć potomków.

Intrinsics dostarcza mechanizmy do zarządzania, konstrukcji i interakcji między obiektami typu composite, ich potomkami i innymi klientami. Zadaniem programisty jest zbudowanie odpowiedniego drzewa, wykorzystując gotowy zbiór obiektów widget. Dodatkowo programista ma możliwość weryfikacji klasy widget w ramach własnych potrzeb. Istnieją mechanizmy, które pozwalają zamienić klasę predefiniowaną na klasę zdefiniowaną przez programistę. Używa się ich wówczas, gdy istnieje potrzeba zamiany w dowolnej klasie danego pola na inne. Wymaga to jednak od programisty dużej wiedzy i wprawy w programowaniu w XToolkit. Mechanizmy pozwalające na zdefiniowanie własnych obiektów widget w sposób szczegółowy są opisane w oryginalnej dokumentacji XToolkit'a.

4.6. Zasoby

Zasobem jest pole w rekordzie widget mające odpowiadające mu wejście (pole) w liście zasobów obiektu widget lub jakiejś jego superklasie. Polu przypisywana jest wartość przez funkcję tworzącą obiekt **XtCreateWidget** (przez nazwanie pola w liście argumentów), przez pole w pliku domyślnych zasobów (bazie danych zasobów), z wewnętrznych wartości domyślnych i przez funkcję **XtSetValues**. Nie wszystkie pola w rekordzie widget są zasobami. Kolejność uzyskiwania wartości zasobów jest następująca:
- z listy argumentów,
- z bazy danych zasobów nie wyszczególnionych w liście argumentów,
- z wewnętrznych wartości domyślnych (jeśli takie istnieją).

Przykładowe wartości w CorePart ustawiane przy inicjalizacji wg listy zasobów Core są następujące:

self	adres struktury widget (nie może być zmieniony)
widget_class	argument **widget_class** dla funkcji **XtCreateWidget** (nie może być zmieniony)
parent	argument parent dla funkcji **XtCreateWidget** (nie może być zmieniony)
xrm_name	zakodowany argument **name** dla **XtCreateWidget** (nie może być zmieniony)
being_destroyed	wartość pola rodzica **being_destroyed**
destroy_callbacks	NULL
constrains	NULL
x	0
y	0
width	0
height	0
border_width	1
managed	**False**
sensitive	**True**
ancestor_sensitive	wynik iloczynu bitowego pól rodzica **sensitive** i **ancestor_sensitive**
event_table	inicjowany przez zarządcę zdarzeń
tm	inicjowany przez zarządcę translacji
accelerators	NULL
border_pixel	**XtDefaultForeground**
border_pixmap	**WtDefaultPixmap**
popup_list	NULL
num_popups	0
name	argument **name** dla **XtCreateWidget** (nie może być zmieniony)
screen	ekran rodzica otrzymany ze specyfikacji dla wyświetlacza
colormap	kopiowany od rodzica podczas tworzenia
window	NULL
depth	poziom zagłębienia
background_pixel	**XtDefaultBackground**
background_pixmap	**XtUnspecifiedPixmap**
visible	**True**
mapped_when_managed	**True**

4.7. Rekordy rozszerzeń klas

Czasami może zaistnieć potrzeba dodania nowych pól do już istniejących struktur klas widget. Aby uniknąć potrzeby rekompilacji wszystkich podklas, ostatnie pole rekordu opisującego klasę jest wskaźnikiem rekordu rozszerzenia. Jeśli nie zdefiniowano żadnego rozszerzenia, podklasa powinna inicjować to pole wartością NULL. Jeśli pole rozszerzeń istnieje, jak to jest w przypadku klas **Composite**, **Constraint** i **Shell**, podklasy mogą dostarczyć wartości dla tych pól, przez ustawienie wskaźnika na odpowiednią część swej struktury klasy (wskaźnik na statycznie zadeklarowany rekord rozszerzeń zawierający dodatkowe pola).

W celu umożliwienia skorzystania z pola rozszerzeń przez wiele podklas rekordy rozszerzeń powinny być deklarowane w formie łączonej listy. Definicja rekordu rozszerzenia powinna zawierać na początku następujące cztery pola:

```
struct{
  XtPointer     next_extention;
  XrmQuark      record_type;
  long          version;
  Cardinal      record_size;
};
```

gdzie:

next_extention - specyfikuje następny rekord rozszerzeń w liście, lub zawiera wartość NULL,

rekord_type - pole identyfikujace kontekst rekordu i używane przez właściciela do lokalizacji konkretnego rekordu w liście rekordów,

version - zdefiniowana przez właściciela stała, która może być użyta do identyfikacji zbiorów binarnych, skompilowanych z alternatywnymi definicjami dla struktur danych rekordów rozszerzeń. Prywatne zbiory nagłówkowe powinny zawierać definicję stałej symbolicznej do identyfikacji tego pola,

record_size - rozmiar rekordu rozszerzeń wraz z czterema polami wspólnymi.

4.8. Klasa "Composite widget"

Obiekty widget będące podklasą klasy **compositeWidgetsClass** mogą mieć arbitralnie ustaloną liczbę potomków i w konsekwencji są odpowiedzialne za więcej działań niż proste obiekty widget. Do ich zadań (zaimplementowanych bezpośrednio w klasie widget lub pośrednio za pomocą funkcji Intrinsic) należy:

- zarządzanie potomkami od ich stworzenia aż do momentu likwidacji,
- likwidacja potomków, gdy jest niszczony obiekt widget,
- zarządzanie strukturą swoich potomków,
- mapowanie (odwzorowanie) zarządzanych potomków.

Za kompleksowe zarządzanie odpowiedzialne są procedury **XtCreateWidget** i **XtDestroyWidget**. **XtCreateWidget** dodaje potomków do rodziców. **XtDestroyWidget**

usuwa potomków. **XtDestroyWidget** zapewnia, że wszyscy potomkowie likwidowanego obiektu także zostaną zlikwidowani.

Okna, należące do zbioru okien, które można wyświetlić, nazywają się oknami zarządzanymi i podlegają zarządcy geometrii. Pozostali potomkowie nazywają się oknami niezarządzanymi i z definicji nie są mapowane.

Do dodawania i usuwania potomków ze zbioru zarządzanych potomków służą funkcje **XtManageChild** i **XtManageChildren** oraz **XtUnmanageChild** i **XtUnmanageChildren**, które zawiadamiają rodziców o potrzebie ponownego przeliczenia układu fizycznego potomków.

4.9. Klasa "Shell widget"

Obiekty widget negocjują swój rozmiar i pozycję ze swoimi rodzicami, to jest obiektami, które bezpośrednio zawierają je. Obiekt na szczycie hierarchii nie ma rodziców i musi współpracować z całym zewnętrznym otoczeniem. Aby to zapewnić, każdy szczytowy obiekt jest hermetyzowany specjalnym obiektem widget shell. Obiekt "shell widget" jest podklasą klasy Composite i to on pozwala na uchylanie się od geometrycznych nakazów w relacjach okien pomiędzy rodzicami i potomkami. Shell może także spełniać rolę warstwy komunikacyjnej z zarządcą okien (ang. window manager). Wyróżnia się siedem typów shell-i, przy czym cztery z nich są obiektami publicznymi. Pozostałe, to shelle wewnętrzne.

1. **Shell**
   Spełnia rolę bazową dla pozostałych shelli.
2. **Override Shell**
   Shell okien używany w celu ominięcia zarządcy okien (np. przy umieszczaniu menu).
3. **WMShell**
   Zawiera pola określające sposób komunikacji między oknami i podklasami shelli.
4. **VendorShell**
   Zawiera pola używane przez specyficzne okna vendor.
5. **Transient Shell**
   Shell okien używany w celu chwilowego wejścia w posiadanie okien, aby później ponownie zwrócić sterowanie nimi zarządcy okien.
6. **TopLevelShell**
   Shell używany na najwyższym poziomie okien.
7. **Application Shell**
   Shell używany przez zarządcę okien do zdefiniowania oddzielnej aplikacji, która będzie aplikacją na najwyższym poziomie.

W XToolkit wyróżnione zostały następujące cztery publiczne shelle:

1. **Override Shell**
2. **Transient Shell**
3. **TopLevelShell**
4. **Application Shell**

Jeżeli obiekt widget typu shell jest skalowany (typowe dla zarządcy okien), to jego potomkowie są również automatycznie skalowani. Podobnie, jeżeli potomek shell'a chce

zmienić rozmiar, to może wywołać żądanie do swojego shell'a. Wynegocjuje on ten rozmiar z zewnętrznym środowiskiem. Klient nigdy nie powinien bezpośrednio zmieniać rozmiaru shell'a.

4.10. Obiekty "pop-up widget"

Obiekty "pop-up widget" są używane do tworzenia okien, które nie należą do hierarchii definiowanej przez drzewo obiektów widget. Każdy potomek pop-up ma okno, które jest potomkiem okna głównego (korzenia). Okna pop-up nie może przysłonić okno rodzica obiektu pop-up. Klasa Composite może posiadać potomków normalnych, jak i typu pop-up.

5. Modyfikacja bibliotek funkcji systemu X - biblioteka Intrinsics

5.1. Inicjalizacja XToolkit'a

Zanim aplikacja może wywołać jakąkolwiek funkcję Intinsics, musi zainicjalizować XToolkit'a używając funkcji:
- XtToolkitInitialize,
- XtCreateApplicationContext,
- XtDisplayInitialize lub XtOpenDisplay,
- XtAppCreateShell

lub zastępującą powyższe funkcje XtAppInitialize.

a) **XtToolkitInitialize** - funkcja inicjuje struktury wewnętrzne XToolkit'a

```
void XtToolkitInitialize ();
```

b) **XtCreateApplicationContext** - funkcja inicjalizuje kontekst aplikacji

Równolegle uruchomionych może być wiele aplikacji XToolkit'a. Każda musi być zdolna do odczytywania danych wejściowych (operacji wejścia) i obsługi zdarzeń niezależnie. Ponadto aplikacje mogą potrzebować wielu połączeń do serwerów systemu X Windows lub mieć obiekty widget na kilku ekranach. Aby sprostać tym wymaganiom, Intinsics definiuje konteksty aplikacji, które pozwalają rozróżniać aplikacje. Głównym komponentem kontekstu aplikacji jest lista wskaźników X Display (X wyświetlaczy) dla tej aplikacji. Każda aplikacja musi posiadać co najmniej jeden kontekst. Do tworzenia kontekstu aplikacji służy funkcja **XtCreateApplicationContext**.

```
XtAppContext XtCreateApplicationContext ();
```

Do usunięcia kontekstu aplikacji i zamknięcia wyświetlaczy służy funkcja **XtDestroyApplicationContext**.

```
void XtDestroyApplicationContext (app_context);
XtAppContext app_context;      // specyfikacja kontekstu
```

Funkcja nie niszczy kontekstu aplikacji, dopóki nie będzie to bezpieczne (tzn. wykonywana dotychczas jakakolwiek usługa nie zostanie doprowadzona do końca). Aby otrzymać kontekst danego obiektu widget, należy użyć **XWidgetToApplicationContext**.

```
XtAppContext XtWidgetToApplicationContext (w);
Widget w;      // specyfikacja obiektu widget, któremu chcemy nadać kontekst
               // aplikacyjny
```

Funkcja zwraca kontekst aplikacji specyfikowanego obiektu.

c) **XtDisplayInitialize** lub **XtOpenDisplay**

Obie funkcje inicjują stan wyświetlacza i dodają go do kontekstu aplikacji.

```
void XtDisplayInitialize (app_context, display, application_name,
    application_class, options, num_options, argc, argv)
Display *XtOpenDisplay (app_context, display_string, application_name,
    application_class, options, num_options, argc, argv);
XtAppContext    app_context;     // specyfikacja kontekstu aplikacyjnego
String          display;         // specyfikacja wyświetlacza
String          display_string;  // specyfikacja łańcucha wyświetlacza lub NULL
String          application_name; // specyfikacja nazwy aplikacji lub NULL w
                                 // wypadku, gdy domyślnie ma to być bieżąca
                                 // aplikacja
String          application_class; // specyfikacja nazwy klasy, od której dziedziczy
                                 // aplikacja
XrmOptionDescRec  *options       // argument option jako parametr
                                 // XrmParseCommand
Cardinal        num_options;     // specyfikacja numeru wejść do listy opcji
Cardinal        *argc;           // wskaźnik na numer z parametrami linii
                                 // komend
String          *argv;           // wskaźnik na argumenty linii komend
```

Jeżeli wartość **display_string** jest równa NULL, to **XtOpenDisplay** używa bieżących wartości opcji **-display** wyspecyfikowanych w **argv**. Jeżeli brak parametrów w **argv**, używa domyślnej wartości wyświetlania (zmiennej środowiskowej DISPLAY). **XtOpenDisplay** zapewnia wygodne programowanie aplikacji. Do zamknięcia wyświetlacza i usunięcia go z kontekstu aplikacji służy **XtCloseDisplay**.

```
void XtCloseDisplay (display );
Display *display;
```

Podobnie jak **XtDestroyApplicationContext** funkcja **XtCloseDisplay** jest bezpieczna w działaniu. **XtCloseDisplay** należy używać, gdy aplikacja po zamknięciu wyświetlacza nadal kontynuuje pracę. W innym wypadku użyjemy **XtDestroyApplicationContext**.

d) **XtAppCreateShell**

Funkcja tworzy początkowy obiekt widget, będący rodzicem dla pozostałych. Jej opis znajduje się w dalszej części.

e) **XtAppInitialize**

Funkcja łączy w sobie wszystkie wyżej wymienione funkcje. Jest łatwym sposobem inicjalizacji XToolkit'a. Do inicjalizacji wewnętrznych struktur XToolkit'a, stworzenia kontekstu aplikacji, otwarcia i zainicjowania wyświetlacza oraz stworzenia początkowego egzemplarza obiektu shell do aplikacji można użyć **XtAppInitialize**.

```
Widget XtAppInitialize (app_context_return, application_class, options,
    num_options, argc_in_out, argv_in_out, fallback_resources, args, num_args)
XtAppContext *app_context_return;   // zwraca kontekst aplikacji, jeżeli jest różny
                                    // od NULL
String       application_class;     // specyfikacja nazwy klasy aplikacji
XrmOptionDescList  options;         // specyfikacja tablicy opcji linii komend
Cardinal     num_options;           // specyfikacja liczby opcji
Cardinal     *argc_in_out;          // wskaźnik na liczbę argumentów linii
                                    // komend
String       *argv_in_out;          // wskaźnik na argumenty linii komend
String       *fallback_resources;   // specyfikacja wartości zasobów, które mają
                                    // być używane przez aplikację, jeżeli plik
                                    // zasobów dla aplikacji nie może być
                                    //otwarty lub odczytany, bądź ma wartość
                                    // NULL
ArgList      args;                  // specyfikacja wartości zasobów, które
                                    // pokryją wartości wyspecyfikowane
                                    // podczas tworzenia obiektu widget shell
Cardinal     num_args               // specyfikacja liczby argumentów
```

Funkcja **XtAppInitialize** wywołuje **XtToolkitInitialize** i **XtCreateApplicationContext**, która to z kolei wywołuje **XtOpenDisplay** z argumentem **display_string** równym NULL i ostatecznie wywołuje **XtAppCreateShell** z argumentem **apllicattion_name** równym NULL, argumentem **widget_class** równym **applicationShellWidgetClass**, argumentami **args** i **num_args**, i zwraca utworzony obiekt shell. Zmodyfikowane argc i argv zwracane przez **XtDisplayInitialize** są zwracane w **argc_in_out** i **argv_in_out**. Jeśli **app_context_return** jest różny od NULL, to stworzony kontekst aplikacji jest także zwracany. Jeśli wyspecyfikowany w linii komend wyświetlacz nie może być otworzony, to generowana jest wiadomość błędu i funkcja **XtAppInitialize** przerywa działanie aplikacji.

5.2. Tworzenie obiektów

Aby utworzyć egzemplarz obiektu widget, należy użyć funkcji **XtCreateWidget**.

```
Widget XtCreateWidget (name, object_class, parent, args, num_args);
String      name;       // nazwa zasobu dla tworzonego obiektu widget (unikalna
                        // dla obiektu tego samego rodzica)
Widget      object_class; // specyfikacja wskaźnika na klasę widget dla tworzonego
                        // obiektu
Widget      parent;     // specyfikacja rodzica obiektu widget
ArgList     args;       // specyfikacja listy argumentów przesłaniających wszystkie
                        // pozostałe zasoby
Cardinal    num_args;   // specyfikacja liczby argementów w liście argumentów
```

Aplikacja może mieć wiele obiektów widget najwyższego poziomu, które potencjalnie mogą być na wielu różnych ekranach. Jeśli aplikacja potrzebuje wielu niezależnych okien, to używa funkcji **XtAppCreateShell**, która tworzy obiekt widget najwyższego poziomu, będący korzeniem drzewa obiektów.

```
Widget XtAppCreateShell ( application_name, application_class, widget_class,
    display, args, num_args)
String      application_name;  // specyfikuje nazwę egzemplarza (jeśli jest
                               // równa NULL, to użyta zostaje nazwa
                               // przekazana do XtDisplayInitialize)
String      application_class; // specyfikuje nazwę klasy dla tej aplikacji
WidgetClass widget_class;      // specyfikuje klasę widget dla obiektu widget
                               // najwyższego poziomu (normalnie jest to
                               // applicationShellWidgetClass)
Display     *display;          // specyfikuje wyświetlacz
ArgList     args;              // lista argumentów dla wartości zasobów
Cardinal    num_args;          // liczba argumentów w liście
```

5.3. Realizacja obiektów widget

Ostatnim etapem konstrukcji instancji obiektu widget jest wykonanie pewnych niezbędnych, dodatkowych działań na obiekcie (m.in. utworzenie X okna, zamapowanie okna obiektu i ewentualnie okien jego potomków), umożliwiających wyświetlenie go. Działania te wykonuje funkcja **XtRealizeWidget**.

```
void XtRealizeWidget (w);
Widget w;       // specyfikacja obiektu widget
```

Jeżeli wyspecyfikowany obiekt widget w już istnieje i jest zrealizowany, to funkcja XtRealizeWidget po prostu zwraca sterowanie.

Wszystkie funkcje Intrinsics i procedury obiektów widget akceptują albo zrealizowane, albo niezrealizowane obiekty widget. Aby sprawdzić, czy obiekt widget został zrealizowany, należy użyć funkcji **XtIsRealized**.

Do stworzenia okna dla danego obiektu widget służy funkcja **XtCreateWindow**, której działanie jest analogiczne do działania funkcji **XCreateWindow** należącej do biblioteki Xlib.

```
void XtCreateWindow(w,window_class,visual,value_mask,attributes)
Widgets              w;            // specyfikacja obiektu widget
unsigned int         window_class; // specyfikacja klasy okna np.InputOutput,
                                   // InputOnly lub CopyFromParent
Visual               *visual;      // typ wizualizacji (zwykle
                                   // CopyFromParent)
XtValueMask          value_mask;   // atrybuty pól do użycia
XSetWindowAttributes attributes;   // atrybuty okna używane przy wywołaniu
                                   // XCreateWindow
```

XtCreateWindow pobiera wartości następujących pól ze struktury obiektu widget - Core:
- depth (poziom zagłębienia),
- screen (monitor),
- parent -> core. window,
- x,
- y,
- width (szerokość okna),
- height (wysokość okna),
- border_width (grubość ramki).

Do usunięcia okien łącznie z obiektem widget służy funkcja **XtUnrealizeWidget.**

```
void XtUnrealizeWidget (w)
Widget w;
```

Jeżeli obsługa obiektu widget właśnie została zakończona, to funkcja **XtUnrealizeWidget** nie wykonuje żadnego działania. W pozostałych wypadkach:
- przerywa zarządzanie obiektem widget,
- odcina potomka od drzewa realizując powrót do rodzica,
- niszczy okna w sposób identyczny jak XtDestroyWindow.

Wszystkie zdarzenia z kolejki zostają usunięte po wywołaniu funkcji **XtUnrealizeWidget**.

5.4. Niszczenie obiektów widget

Intrinsics zapewnia:
- zniszczenie wszystkich potomków pop-up, z obiektu widget, który jest niszczony,
- odmapowanie obiektu widget od swojego rodzica,

- wywołanie procedury powrotnej, która zostanie zamapowana w momencie, gdy obiekt widget zostanie zniszczony,
- dealokację w momencie niszczenia.

Do zniszczenia obiektu widget służy funkcja **XtDestroyWidget**.

```
void XtDestroyWidget (w)
Widget w;
```

Funkcja **XtDestroyWidget** niszczy tylko metody danego obiektu, którego dotyczy, zaś pozostałe obiekty należące do danego obiektu widget muszą swoje metody zniszczyć same.

5.5. Zarządzanie potomkami obiektów "composite widget"

Aby dodać listę obiektów widget do zbioru geometrycznie zarządzanych (a więc potencjalnie wyświetlanych) obiektów rodzica obiektu "composite.widget", należy najpierw stworzyć nowe obiekty (**XtCreateWidget**) i następnie wywołać **XtManageChildren**.

```
typedef Widget *WidgetList
void XtManageChildren (children, num_children)
WidgetList children;        // specyfikacja listy potomków obiektu
Cardinal   num_children;    // specyfikacja liczby potomków
```

Aby dodać pojedynczego potomka do listy zarządzanych potomków, należy najpierw stworzyć obiekt widget (**XtCreateWidget**), a następnie użyć funkcji **XtManageChild**.

```
void XtManageChild (child)
Widget child;               //specyfikacja potomka
```

Funkcja ta konstruuje **WidgetList** o długości równej jeden i wywołuje **XtManageChildren**.

Do stworzenia potomka i zarządzania nim służy **XtCreateManagedWidget**.

```
Widget XtCreateManagedWidget (name, widget_class, parent,args, num_args)
String      name;          // nazwa tworzonego obiektu widget
WidgetClass widget_class;  // wskaźnik na klasę tworzonego obiektu widget
Widget      parent;        // rodzic obiektu widget
ArgList     args;          // lista argumentów, które mają pierwszeństwo nad
                           // pozostałymi specyfikacjami zasobów
Cardinal    num_args;      // specyfikacja liczby argumentów listy
```

Funkcja ta jest wygodniejsza w użyciu niż **XtCreateWidget** i **XtManageChild**.

Aby usunąć kilku potomków z listy zarządzanych potomków danego obiektu widget, należy użyć **XtUnmanageChildren**.

```
void XtUnmanagedChildren (children, num_children)

WidgetList    children;        // specyfikacja listy dzieci
Cardinal      num_children;    //specyfikacja liczby dzieci
```

Funkcja **XtUnmanageChildren** nie niszczy obiektów potomnych, a usuwa je jedynie z kolejki zarządzanych potomków. Zatem ten sam obiekt widget może zostać później do niej ponownie wstawiony. Tę samą czynność w stosunku do pojedynczego dziecka wykonuje funkcja **XtUnmanageChild**

```
void XtUnmanageChild (child)

Widget    child;        // specyfikacja potomka
```

Funkcja ta konstruuje **WidgetList** o długości równej jeden i wywołuje **XtUnmanageChildren**.

5.6. Tworzenie obiektów "pop-up shell"

Do tworzenia potomka pop-up należy zawsze używać podklasy shell. Nie wolno tworzyć potomków pop-up od innych rodziców. Jednocześnie możliwe jest zarządzanie tylko jednym potomkiem. Gdy należy użyć obiektu pop-up, trzeba odwołać się do niego nie bezpośrednio, lecz przez shella. Do stworzenia obiektu pop-up shella służy funkcja **XtCreatePopupShell**.

```
Widget XtCreatePopupShell (name, widget_class, parent, args, num_args)

String        name;          // nazwa tworzonego shella
WidgetClass   widget_class;  // specyfikacja wskaźnika klasy dla tworzonego
                             // obiektu widget shell
Widget        parent;        // specyfikacja rodzica obiektu widget
ArgList       args;          // lista argumentów mających pierwszeństwo nad
                             // pozostałymi wyspecyfikowanymi źródłami
Cardinal      num_args;      // liczba argumentów na liście argumentów
```

Funkcja ta zapewnia powstanie podklasy klasy shell i dołączenie potomka bezpośrednio do listy rodziców. Obiekt pop-up może być włączany przez różne mechanizmy:
- wywołanie **XtPopup**,
- jedną z procedur **XtCallbackNone**, **XtCallbackNoneExclusive** lub **XtCallbackExclusive**,
- standardową funkcją **XtMenuPopup**. Do mapowania pop-up z dowolnego miejsca aplikacji używa się **XtPopup**.

```
void XtPopup (popup_shell, grab_kind)

Widget        popup_shell; // specyfikacja obiektu pop-up widget;
XtGrabKind    grab_kind;   // specyfikacja sposobu, jakim zdarzenie będzie wymuszone
```

Do mapowania menu w momencie, kiedy przycisk myszy jest wciśnięty i jej wskaźnik znajduje się w oknie danego obiektu widget, służy funkcja **XtMenuPopup**.

```
void XtMenuPopup (shell_name)
String  shell_name;  // specyfikacja obiektu widget typu pop-up shell
```

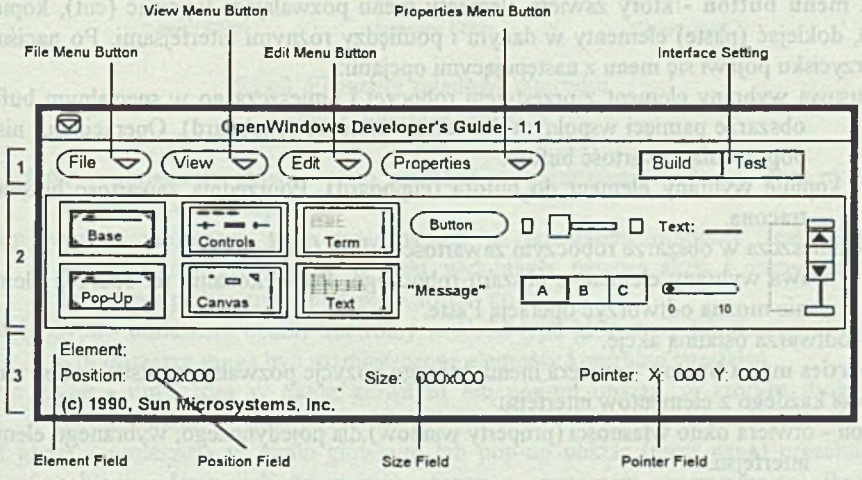
Wymapowywanie może się odbywać na poniższe sposoby:
- wywołanie **XtPopdown,**
- wywołanie **XtCallbackPopdown,**
- wywołanie **XtMenuPopdown.**

5.7. Wyjście z aplikacji

Wszystkie aplikacje XToolkit powinny być zamykane przez wywołanie funkcji **XtDestroyApplicationContext** i powrót do systemu operacyjnego. Inną metodą, dającą taki sam efekt, jest odmapowywanie na każdym poziomie widget-shella, poprzez wywołanie **XtUnmapWidget** i zamknięcie okien. Należy pamiętać, że przy powrocie z aplikacji należy zwolnić przyłączone zasoby jedną z powyższych metod.

6. Budowa aplikacji z użyciem systemu Devguide

Devguide jest narzędziem programisty wspomagającym tworzenie interfejsu graficznego aplikacji z użytkownikiem (GUI). Umożliwia budowę, modyfikację, testowanie i generację kodu źródłowego w języku C, zaprojektowanego GUI. Wykorzystanie tego narzędzia usprawnia, w stosunku do wykorzystania takich narzędzi jak XToolkit, budowę aplikacji komunikujących się z użytkownikiem z wykorzystaniem GUI w systemie SunOS.



Rys.2. Główne okno systemu Devguide
Fig.2. Devguide - main screen

6.1. Opis elementów systemu

Przestrzeń okna podzielona jest na trzy obszary, oznaczone na rysunku 2 odpowiednio numerami:

1. **CONTROLS** - przestrzeń kontroli, służy do sterowania przebiegiem budowy interfejsu. Składa się z następujących elementów:

• **File menu button** - zawiera pozycje menu, za pomocą których można ładować, zamykać i zapisywać cały, bądź fragment tworzonego interfejsu. Wywołanie menu następuje po wciśnięciu prawego skrajnego klawisza myszy i udostępnia następujące opcje:

Load - ładuje wcześniej zapisany interfejs. Po wybraniu tej opcji pojawi się okno, w którym należy podać ścieżkę dostępu (pole Directory:) i nazwę interfejsu (pole Interface Name:). Wciśnięcie przycisku (button) "Load" spowoduje załadowanie podanego interfejsu i pojawienie się go w przestrzeni roboczej (jeśli ścieżka dostępu i jego nazwa są podane poprawnie). Jednocześnie może być załadowany tylko jeden interfejs.

Save - zapisuje budowany właśnie interfejs do pliku z rozszerzeniem .G. Interfejs musi być wcześniej zapisany, tzn. mieć nadaną nazwę.

Save As - zapisuje nowo utworzony interfejs, zapisuje interfejs z nową nazwą lub zapisuje część interfejsu. Po wybraniu tej opcji pojawi się okno, w którym należy podać kartotekę (Directory:) i nazwę zbioru, w którym zapisany będzie interfejs (Interface Name:). Nazwa zbioru musi spełniać konwencję systemu operacyjnego (SunOS). Pamiętać należy, że Devguide dodaje rozszerzenie .G do podanej nazwy.

Close - zamyka interfejs będący w przestrzeni roboczej i usuwa wszystkie jego elementy składowe. Przy próbie usunięcia nie zapisanych zmian pojawi się ostrzeżenie.

• **View menu button** - który otwiera poprzednio zwolnione okna typu pop-up (dismissed windows) operacją "dismiss". Opcja ta nie otwiera okna głównego ani nie odtwarza skasowanego (delete).

• **Edit menu button** - który zawiera elementy menu pozwalające wycinać (cut), kopiować (copy), doklejać (paste) elementy w danym i pomiędzy różnymi interfejsami. Po naciśnięciu tego przycisku pojawi się menu z następującymi opcjami:

Cut - usuwa wybrany element z przestrzeni roboczej i umieszcza go w specjalnym buforze, obszarze pamięci wspólnym dla całego systemu (clipboard). Operacja cut niszczy poprzednią zawartość bufora.

Copy - kopiuje wybrany element do bufora (clipboard). Poprzednia zawartość bufora jest tracona.

Paste - umieszcza w obszarze roboczym zawartość bufora (clipboard).

Delete - usuwa wybrany element z obszaru roboczego. Po wykonaniu tej operacji elementu nie można odtworzyć operacją Paste.

Undo - odtwarza ostatnią akcję.

• **Properties menu button** - otwiera menu, którego pozycje pozwalają na ustawienie i zmianę własności każdego z elementów interfejsu.

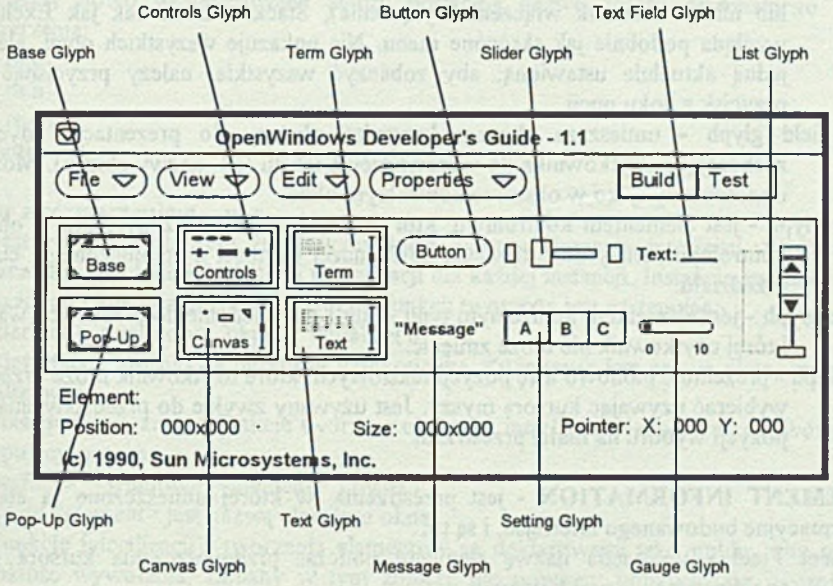
Selection - otwiera okno własności (property window) dla pojedynczego, wybranego elementu interfejsu.

Pozostałe pozycje menu otwierają odpowiednie okna własności dla każdego elementu. Są to: Base Windows, Pop-up Windows, Control Areas, Canvas Panes, Term Panes, Text Panes,

Buttons, Messages, Settings, Text Fields, Sliders, Guages, Lists, Help (otwiera okno edytora, w którym można wprowadzić tekst pomocy dla danego elementu interfejsu), Menus (otwiera okno służące do tworzenia i zmiany menu).

• Interface setting - służy do zmiany trybu pracy systemu: tryb "Build" jest przeznaczony do budowania interfejsu: w trybie tym można dodawać lub usuwać elementy oraz zmieniać ich własności. Tryb: "Test" - do testowania współpracy elementów. Aplikacja nie wykonuje żadnych funkcji, gdyż mogą być one przypisane dopiero po utworzeniu kodu źródłowego za pomocą gxv. Pozwala jedynie zobaczyć, jak zachowują się poszczególne elementy interfejsu.

2. ELEMENT GLYPHS - zawiera elementy, z których można budować interfejs. Rysunek 3 zawiera położenie i nazwy poszczególnych elementów.



Rys.3. Elementy składowe interfejsu
Fig.3. Interface elements

• Base glyph - umieszcza główne okno aplikacji w przestrzeni roboczej. Jest to okno podstawowe aplikacji.

• Pop-up glyph - umieszcza okno typu pop-up w przestrzeni roboczej. Jest to okno pomocnicze wywoływane w celu wykonania pewnej akcji i zamykane (np. ustawienie parametrów, podanie wartości itp.).

• Controls glyph - umieszcza obszar kontrolny w oknie typu głównego, lub pop-up. Tylko w tym obszarze mogą być rozmieszczone elementy kontrolne interfejsu.

• Canvas glyph - umieszcza w oknie głównym lub pop-up obszar, w którym mogą być rozmieszczone elementy rysunków.

• Term glyph - umieszcza w oknie głównym lub pop-up obszar (term pane) prezentujący nakładkę Unix-a i stanowiący sprzęg z systemem operacyjnym. Pozwala użytkownikowi wydawać komendy systemowe oraz przewijać zawartość okna w tył i przód.

- Text pane glyph - umieszcza obszar tekstu (text pane) w oknie głównym i pop-up. Jest to obszar okna, gdzie użytkownik może przeglądać i wprowadzać swój tekst. Obszar ten wyposażony jest w specjalny element kontrolny pozwalający przewijać tekst w oknie (scrollbar).
- Button glyph - umieszcza przycisk (button) w obszarze kontrolnym (controls glyph) okna, który może także prezentować menu.
- Message glyph - umieszcza wiadomość tekstową dla użytkownika w obszarze kontrolnym okna.
- Setting glyph - jest elementem kontrolnym, który umożliwia wybór i ustawienie dwustanowe zestawu opcji. Możliwe są cztery rodzaje pracy: Exclusive - tylko jedna opcja jest wybrana, Nonexclusive - możliwe jest ustawienie kilku opcji jednocześnie, Check Box - działa tak jak Nonexclusive, ale różni się sposobem prezentacji (pojawia się lub niknie znacznik włączenia/wyłączenia), Stack - działa tak jak Exclusive i wygląda podobnie jak skrócone menu. Nie pokazuje wszystkich opcji, ale tylko jedną aktualnie ustawioną; aby zobaczyć wszystkie, należy przycisnąć mały przycisk z boku opcji.
- Text field glyph - umieszcza element kontrolny służący do prezentacji, edycji lub zachęcający użytkownika do wprowadzenia tekstu (np. nazwy zbioru). Może być umieszczony tylko w obszarze kontrolnym okna.
- Slider glyph - jest elementem kontrolnym, który może być umieszczony tylko w obszarze kontrolnym, służy do ustawiania pojedynczej wartości z ograniczonego, ciągłego przedziału.
- Gauge glyph - jest elementem kontrolnym typu slider i służy jedynie do prezentacji wartości, której użytkownik nie może zmienić.
- List glyph - prezentuje pionowo listę pozycji tekstowych, które użytkownik może przewijać i wybierać używając kursora myszy. Jest używany zwykle do przedstawienia wielu pozycji wyboru na małej przestrzeni.

3. **ELEMENT INFORMATION** - jest przestrzenią, w której umieszczone są elementy informacyjne budowanego interfejsu, i są to:
- Element Field - wyświetla nazwę elementu podczas przemieszczania kursora myszy znajdującego się pod kursorem.
- Position Field - wyświetla względną pozycję elementu w punktach.
- Size Field - wyświetla wielkość elementu w punktach ekranu.
- Pointer Field - wyświetla względną pozycję kursora myszy w punktach ekranu.

6.2. Tworzenie interfejsu graficznego

Pierwszym etapem tworzenia aplikacji jest faza projektowania interfejsu graficznego poprzez zaprojektowanie okien, menu oraz ich współdziałania. Do tego celu służy właśnie program Devguide. Kiedy faza projektowania jest zakończona, należy zapisać utworzony interfejs w specjalnym zbiorze GIL (Graphical Intermediate Language). Zbiór ten zawiera opis elementów interfejsu w pośrednim języku opisowym. Jeśli interfejs składa się z więcej niż jednego okna, zalecane jest tworzenie każdego okna jako oddzielnego interfejsu i zapis w oddzielnych zbiorach GIL oraz jeśli to konieczne użycie funkcji komunikacji pomiędzy oknami. Taki sposób znacznie ułatwia kompilację, gdyż nie występują problemy związane z

dużymi rozmiarami zbiorów. Wymaga to zmodyfikowania zbioru Makefile przy kompilacji i linkowania zbiorów.

6.3. Generowanie kodu źródłowego

Po utworzeniu interfejsu i zapisaniu go w jednym lub kilku zbiorach GIL można przystąpić do generacji kodu źródłowego. Używa się do tego programu "gxv", którego parametrem jest nazwa zbioru GIL (zbiór taki posiada rozszerzenie .G). Gxv tworzy cztery zbiory źródłowe i zbior Makefile, który służy do kompilacji zbiorów źródłowych. Jeśli zbiorów GIL jest więcej, należy je włączyć w zbiorze Makefile.

Cztery nowo wygenerowane zbiory posiadają nazwę zbioru oryginalnego i dodane rozszerzenia:

_ui.c
_ui.h
_stubs.c
.info

Zbiór z rozszerzeniem _ui.c

Jest głównym zbiorem źródłowym, który tworzy elementy interfejsu. Rozpoczyna się deklaracjami #include i funkcjami inicjalizacji dla każdej instancji. Instancja jest inicjalizowana dla każdego okna w zbiorze GIL. Nazwa funkcji tworzona jest następująco:
" <interface>_<window>_objects_initialize()"

<interface> jest nazwą interfejsu użytkownika, <window> jest nazwą okna, które ma być inicjalizowane.

Następnie zawarte są funkcje tworzące elementy interfejsu. Nazwy funkcji tworzone są w następujący sposób:
"<interface>_<window>_<element>_create()",

gdzie <element> jest nazwą elementu okna.

Funkcje inicjalizacji i tworzenia elementów są deklarowane jako public, aby umożliwić dowolność wywołania. Zmiany w tym zbiorze nie powinny odbywać się "ręcznie", gdyż ponowne wywołanie gxv zniszczy to, co zostało zrobione.

Zbiór z rozszerzeniem _ui.h

Zbiór ten zawiera deklaracje extern, definicje typów oraz struktur, które zawierają wskaźniki do okna i jego elementów.

Zbiór z rozszerzeniem _stubs.c

Zbiór ten zawiera schematy funkcji callback-owych dla elementów, które tego wymagają. Zbiór ten zawiera również schemat dla funkcji main() wraz z wywołaniami funkcji inicjalizujących.

Zbiór z rozszerzeniem .info

Zbiór zawiera tekst pomocy, jaki utworzyliśmy dla elementów interfejsu.

Zbiór Makefile

Gdy uruchomimy program gxv, to sprawdzi on, czy zbiór Makefile istnieje, jeśli tak, to nie utworzy nowego, co ma zapobiec utracie poprzedniego zbioru Makefile. Po utworzeniu zbioru Makefile można go zmienić według własnych potrzeb. Początek zbioru zawiera sekcję nazwaną "Parameters", w której znajduje się pięć parametrów:

PROGRAM - zawiera nazwę, jaką otrzyma zbiór wykonywalny podczas kompilacji.

SOURCES.c - zawiera nazwy zbiorów źródłowych w języku C składających się na aplikację.

SOURCES.h - zawiera nazwy zbiorów nagłówkowych z rozszerzeniem .h składających się na aplikację.

SOURCES.g - zawiera nazwy zbiorów GIL składających się na aplikację.

STUBS.G - zawiera nazwy zbiorów GIL dla zmienianych funkcji callback-owych (są to te zbiory, których schematy funkcji nie zostały skopiowane do własnego zbioru z rozszerzeniem .c zawartego w SOURCES.c)

Kiedy Gxv tworzy zbiór Makefile, zbiór ten nie zawiera żadnych nazw zbiorów dla parametrów SOURCES.c i SOURCES.h i należy zrobić to ręcznie.

6.4. Kompilacja

Aby kompilacja przebiegła poprawnie, należy wcześniej poprawnie zdefiniować środowisko pracy. Dotyczy to głównie dwóch zmiennych: GUIDHOME i OPENWINHOME, które powinny zawierać ścieżki dostępu do Devguide i Open Windows. Należy również uzupełnić parametry w zbiorze Makefile.

Następnie należy wydać komendę: "make", która wykona się zgodnie z zawartością zbioru Makefile. Make najpierw sprawdzi, czy zbiory wyspecyfikowane w SOURCES.G się zmieniły, jeśli tak, to użyje gxv do utworzenia nowych zbiorów wynikowych. Operacja zostanie zakończona przez kompilację i konsolidację wszystkich wyspecyfikowanych zbiorów.

Skompilowany kod będzie zawarty w zbiorze, który jest zawarty w parametrze PROGRAM. Aby go uruchomić, wystarczy podać jego nazwę.

6.5. Budowa przykładowej aplikacji

Wygenerowany kod za pomocą Devguide nie wykonuje żadnych innych operacji poza utworzeniem okien, przycisków, menu itp., dlatego konieczne będzie dołączenie kodu, który wykonywałby jakąś akcję i dawał wyniki, które można zaprezentować graficznie. Załóżmy, że kod ten zapisany jest w pliku o nazwie **przyklad.c**.

Po stworzeniu za pomocą myszy odpowiedniego interfejsu graficznego należy:

1. Za pomocą File menu button zapisać go pod nazwą example (powstanie zbiór example.G)
2. Uruchomić program gxv z parametrem example. W ten sposób zostaną utworzone zbiory:

example_ui.c
example_ui.h
example_stubs.c
example.info
oraz Makefile

3.Skopiować zbiór example_stubs.c do nowego o nazwie example.c
4.W zbiorze example.c wypełnić szablony funkcji obsługujących zdarzenia w przyszłej aplikacji tak, aby odwoływały się do kodu zapisanego w pliku **przyklad.c**
5. Uzupełnić w zbiorze Makefile parametr SOURCES.c:
SOURCES.c = example.c **przyklad.c**
6.Uruchomić program make w celu utworzenia aplikacji o nazwie example

7. Podsumowanie

Środowisko systemu X Windows umożliwia budowanie aplikacji spełniających złożone wymagania funkcjonalne dotyczące m.in. integracji systemów, bezpieczeństwa, elastyczności, pracy w sieci komputerowej, przenośności oprogramowania, skalowalności itp. Jest podstawowym elementem w budowie aplikacji graficznych dla systemów otwartych. Bogactwo funkcjonalne ma także swoje odbicie w bogactwie dostępnych narzędzi programistycznych; od kilku warstw bibliotek standardowych, jak opisane w artykule, po narzędzia wysokiego poziomu jak Devguide. Bogactwo funkcjonalne niestety okupione zostało dużą złożonością kodu aplikacji, szczególnie dla bibliotek niskiego poziomu. Stąd istotna wydaje sie być rola narzędzi zwalniających programistę z bezpośredniego kodowania standardowych elementów aplikacji jak Devguide. Dokonany przegląd bibliotek standardowych i narzędzi nie wyczerpuje zagadnienia. Przedstawione narzędzia obejmują raczej dwa skrajne elementy w kostrukcji aplikacji w systemie X Windows: biblioteki niskiego poziomu (Xlib, XToolkit) oraz aplikacje do generacji interfejsu graficznego Devguide.

LITERATURA

[1] SCO Xsight Developement System -X library.
[2] SCO Xsight Developement System -XToolkit.
[3] OpenWindows Developer's Guide 1.1 User's Manual.

Recenzent: Dr hab. inż. Adam Mrózek

Wpłynęło do Redakcji 22 listopada 1994 r.

Abstract

The purpose of this paper is to present general ideas of programming in a network graphical environment called X Windows. In particular the architecture and programming environment were described. We present basic libraries Xlib, XToolkit and a short overview how to use them. In paper there are also described some aspects of proper environment configuration. The DevGuide package for quick developement of X Windows-based aplications is also presented.