Objektorientierte BS — Beispiel Choices

Überblick

❑ Framework-Konzepte

❑ Framework-Dokumentation

❑ Choices-Überblick

❑ Prozeßverwaltung

❑ Speicherverwaltung

❑ Proxy-Objekte (Systemschnittstelle)

Literatur

CaI93 Roy H. Campbell and Nayeem Islam, "CHOICES: A Parallel Object Orientd Operating System", in Gul Agha, Peter Wegner, and Akinori Yonezawa [Eds.]: *Research Directions in Concurrent Object-Oriented Programming,* MIT Press, Cambridge, Massachusetts, 1993.

CIRM93 Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany, "Designing and Implementing CHOICES: An Object-Oriented System in C++", *Communications of the ACM*, Vol. 36, No. 9, pp. 117 - 126, Sep. 1993.

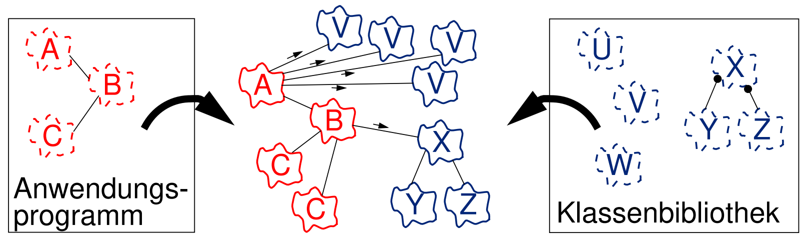
Weitere Literatur im Handapparat der Gruppenbibliothek Informatik:
Ordner zu OOBP, Abschnitt Choices

Frameworks

❑ Klasse: Design für ein Objekt

❑ Klassenbibliothek

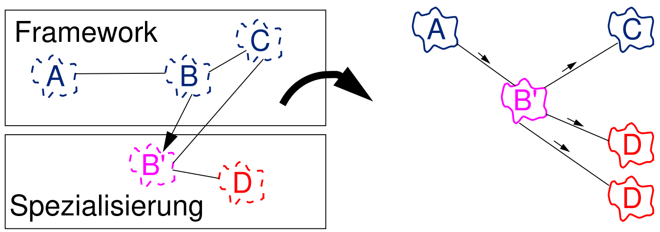
- Sammlung von Klassen
  → Aufbau und Ergänzung von Software
- Klassen in der Bibliothek können voneinander unabhängig sein
- Kontrollfluß: Anwendungsobjekte → Bibliotheksobjekte



Frameworks (2)

❑ Framework:
Design einer Anwendung / einer Anwendungskomponente
- Architekturdesign eines objektorientierten Systems
  - ➜ *High-level Design* der Software — schwierigster Teil der Software-Erstellung
    - – Verteilung der Funktionalität auf die Komponenten
    - – Design von Schnittstellen

❑ Framework: Menge von Klassen
- bilden zusammen ein Anwendungsgerüst
- Komponenten werden vom Entwickler
  – hinzugefügt – ersetzt – modifiziert
- Kontrollfluß:
  Framework → Anwendungsobjekte → Framework
  "**don't call us, we'll call you**"



Frameworks (3)

❑ Framework beschreibt:
- Dekomposition eines Problems
  - – nicht nur Klassen, sondern wie die Klassen zusammenarbeiten
- Zusammenhänge zwischen Komponenten
  - – Constraints & Invarianten
    - Vorgabe von Einschränkungen für die Interaktion von Komponenten
    - gemeinsame Invarianten die von Objekten zu erfüllen sind und wie sie erfüllt werden
  - – Vererbung
    - Vorgabe von Klassenhierarchien (auch Hierarchien abstrakter Klassen)
  - – Inclusion Polymorphism
    - Anpassung von einzelnen Komponenten durch konforme Subtypen
  - – informelle Kompositionsregeln

❑ Ziele von Frameworks
- Aufbau von Anwendungen aus existierenden Komponenten
- Häufige Wiederverwendung einer kleinen Menge von Typen
- So wenig neuer Code wie möglich

➜ **"It shouldn't take a good programmer to build a good program"**

Hierarchische Frameworks

- ❏ Subsystem = funktionale Komponente eines Systems
- ❏ Subframework: Architekturdesign eines Subsystems
  - ➔ Verfeinerung des allgemeinen Frameworks für ein Subsystem
- ❏ Konformität der Subframeworks
  - – Wiederverwendung von Klassen und Constraints des Frameworks
  - – Subframework kann weitere Komponenten und Constraints sowie Unterklassen-Komponenten einführen

Framework-Dokumentation

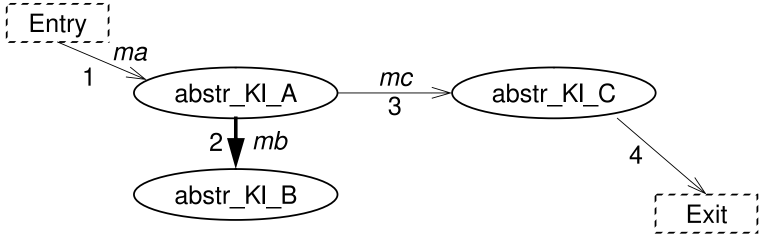
- ❏ Klassenhierarchien

abstrakte_Klasse_A — konkrete_Unterklasse_von_A

abstr_Kl_B — abstr_Unterkl_B1 — 1._konkr_Unterkl_von_B1
abstr_Kl_B — abstr_Unterkl_B1 — 2._konkr_Unterkl_von_B1

Framework-Dokumentation

❑ Kontrollfluß zwischen abstrakten Klassen



einfacher Methodenaufruf

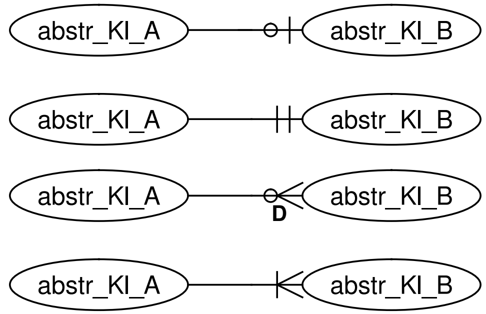
Methodenaufruf & return

Methodenaufrufe in verteilter Umgebung

jeweils mit Aufrufreihenfolge und aufgerufener Methode

Framework-Dokumentation

❑ Beziehungen zwischen abstrakten Klassen



optionale 1:1 Beziehung

vorgeschriebene 1:1 Beziehung

optionale 1:n Beziehung (dynamisch veränderbar)

vorgeschriebene 1:n Beziehung

– **D** markiert eine Beziehung als dynamisch (zur Laufzeit) veränderbar

Framework-Dokumentation

- Verfeinerung von Kontrollfluß-Graphen und Beziehungsbeschreibungen
  - abstrakte Klasse → konkrete Klasse
  - zusätzliche abstrakte und konkrete Klassen
  - zusätzliche Beziehungen und Kontrollflüsse
- Komposition von Kontrollfluß-Beschreibungen
  - Exit- und Entry-Knoten verschiedener Graphen können verbunden werden
  - Graphen, die nicht durch Kanten verbunden sind, sind parallel ausführbar

Choices-Überblick

- Betriebssystem-Framework
- entwickelt seit 1987 an der University of Illinois at Urbana-Champaign
- Implementierung in C++
  - Sun SPARCstation I + II, Encore Multimax, Emulation auf UNIX
  - über 300 Klassen, ca. 150 000 Zeilen Code
- beschreibt Frameworks für
  - Prozeßverwaltung
  - virtuelle Speicherverwaltung
  - Dateisysteme
  - Netzwerk

Choices-Framework

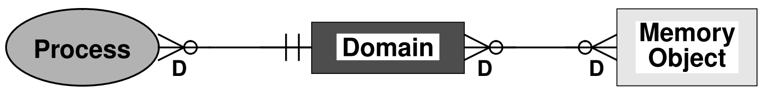
Komponenten

❑ MemoryObject
- Datenspeicher

❑ Process
- Aktivitätsträger zur sequentiellen Ausführung eines Algorithmus

❑ Domain
- Umgebung, die Namen, die von einem Aktivitätsträger zur Benennung von Daten benutzt werden (z. B. virtuelle Adressen) in Speicheradressen abbildet
(ein Domain ist damit eine Abstraktion eines virtuellen Adreßraums)

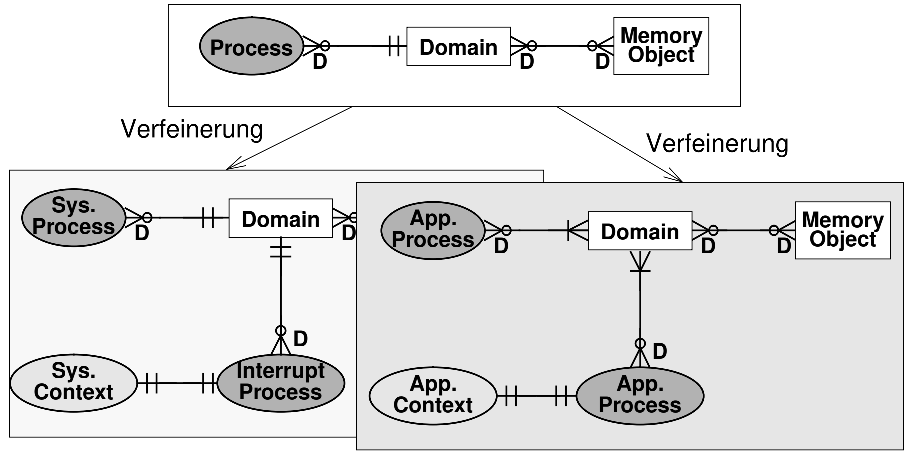
Beziehungen



- Einem Domain sind mehrere Prozesse zugeordnet (➔ leichtgewichtige Prozesse oder Threads). Es können dynamisch weitere Prozesse in einem Domain hinzukommen — d. h. es können neue erzeugt werden.
- Ein Prozeß ist immer genau einem Domain zugeordnet. Die Zuordnung ist nicht dynamisch veränderbar — d. h. Prozesse können nicht zwischen Domains wandern.
- Jedem Domain sind mehrere Memory Objects zugeordnet (das entspricht den Segmenten).
- Ein Memory Object kann mehreren Domains zugeordnet sein (= Shared memory).
- Die Zuordnungen können dynamisch verändert werden. Es können somit neue Memory Objects in einem Domain hinzukommen (= Speicher nachallokieren), Memory Objects aus einem Domain entfernt werden (= Speicher freigeben oder Shared memory deallokieren) und Memory Objects in weiterer Domains abgebildet werden (Shared memory Zuordnungen verändern).

Choices-Framework

Verfeinerung (Beispiel)

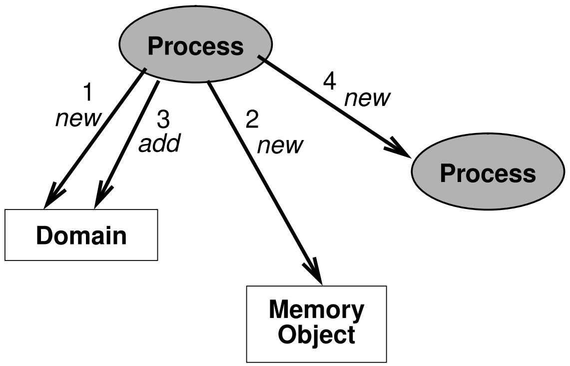


- Die abstrakte Klasse Process wurde durch die konkrete Klasse Application Process ersetzt.
- Die Zuordnung Kontext - Prozeß wurde neu aufgenommen. Ein Kontext-Objekt dient der Speicherung des Prozeßzustands im Fall von Prozeßumschaltungen (d. h. Programmzähler, Registerwerte, etc.).
- Die Zuordnung eines User-Domains und eines Kernel-Domains zu einem Anwendungsprozeß ist durch den Übergang von einer 1:1 zu einer 1:n-Beziehung zwischen Prozeß und Domain beschrieben (ein System-Prozeß dagegen hat nur genau einen Domain zugeordnet, deshalb konnte die 1:n-Beziehung in dem allgemeinen Kontrollfluß noch nicht angegeben werden!).

Choices-Framework

Kontrollfluß (Beispiel)

❑ Erzeugung eines Prozesses



1. neuer Domain wird erzeugt

2. neue MemoryObjects für Datensegment, Textsegment, initialisierte Daten, Stack, etc. werden erzeugt

3. die MemoryObjects werden in den neuen Domain eingetragen

4. der neue Process wird erzeugt; ihm werden der neue Domain, eine Einsprungadresse und initiale Argumente für den Konstruktor übergeben

Prozeßverwaltung

Komponenten

❑ Process

- **Ablaufpfad durch eine Gruppe von C++ -Objekten**.
  - Benutzer-Prozesse laufen im Benutzeradreßraum oder im Kern
  - System-Prozesse laufen im Kern und sind nicht unterbrechbar
  - Interrupt-Prozesse werden zur Bearbeitung von Interrupts benutzt
- Objekte der Klasse Process kapseln die architekturunabhängigen Informationen über einen Prozeß
- jedem Prozeß ist genau ein Domain zugeordnet, in dem er ausgeführt wird
- mehrere Prozesse können gemeinsam in einem Domain ablaufen
- Prozeßumschaltungen zwischen Prozessen eines Domains sind billig (light-weight), Umschaltungen zwischen Prozessen verschiedener Domains sind teuer (heavy-weight)

❑ ProcessorContext

- kapselt **architekturabhängige Informationen über einen Prozeßzustand**
- enthält Methoden zum Sichern und Restaurieren des Prozeßzustands bei einem Prozeßwechsel
- 1:1-Beziehung zwischen einem ProcessorContext und einem Process

❑ Processor

- kapselt die **architekturabhängigen Details einer CPU**

❑ Gang

- Gruppe von Prozessen, die **gemeinsam "gescheduled"** werden sollen, d. h., die als Gruppe parallel auf den Prozessoren eines Multiprozessors abgearbeitet werden sollen
- erlaubt es, Mengen von Prozessen gemeinsam zu bearbeiten

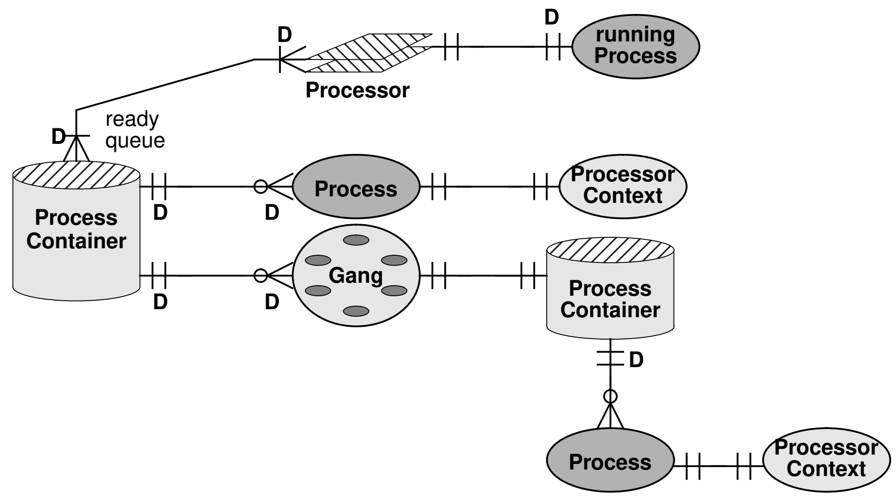
❑ ProcessContainer

- **implementiert Scheduling** in Choices
- Unterklassen implementieren verschiedene Scheduler und Prozeßwarteschlangen

Klassenhierarchien

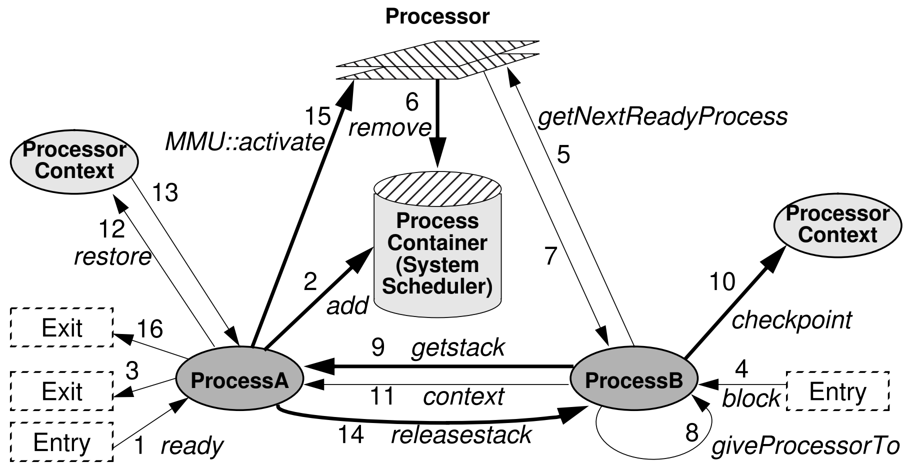
- Process
  - ApplicationProcess
  - SystemProcess
    - PreemptableSystemProcess
  - InterruptProcess
  - Gang
- ProcessorContext
  - ApplicationContext
  - SystemContext
    - UninterruptableSystemContext
- Processor
  - NS323332Processor
  - SPARCProcessor
- ProcessContainer
  - FIFOScheduler
    - RRScheduler
      - RRSchedulerGangs
    - FIFOGangScheduler
  - SystemScheduler

Klassenbeziehungen



- Jedem Prozessor ist genau ein aktuell laufender Prozeß (running process) und eine Prozeßwarteschlange (SystemScheduler, der ProcessContainer für die ready-queue) zugeordnet. Wenn der laufende Prozeß sich blockiert, entnimmt der Prozessor einen neuen Prozeß aus einem ihm zugeordneten ProcessContainer.
- Ein ProcessContainer kann einem oder mehreren Prozessoren zugeordnet sein - die Zuordnung kann dynamisch verändert werden.
- Ein ProcessContainer kann mehrere Prozesse oder Gangs enthalten.
- Jeder Prozeß besitzt genau einen ProcessorContext und ist genau einem ProcessContainer zugeordnet, in den er eingehängt wird, wenn er laufbereit ist. Die Zuordnung eines Prozesses zu einem ProcessContainer kann verändert werden.
- Einer Gang ist ein FIFOGangScheduler zugeordnet sowie ein spezieller ProcessContainer, der die Gang-Mitglieder enthält. Wenn eine Gang über die ready-queue dem Prozessor zugeteilt wird, stellt der SystemScheduler die Anzahl der Gang-Mitglieder fest und weist den Prozessor an, einen aktuellen Gang-Prozeß aus dem FIFOGangScheduler zu entnehmen. Der FIFOGangScheduler ist zu diesem Zeitpunkt leer und der Prozessor wartet aktiv auf Prozesse in dem FIFOGangScheduler. Wenn der SystemScheduler die benötigte Anzahl an Prozessoren allokiert hat, überträgt er Gang-Mitglieder aus dem Gang-ProcessContainer in den FIFOGangScheduler. Die Prozessoren können dann Gang-Mitglieder aus dem FIFOGangScheduler entnehmen und aktivieren.

Prozeßumschaltung



1. Prozeß A wird in Zustand ready versetzt (= ein Aktivitätsträger eines anderen Prozesses ruft die Methode ready an dem Objekt ProcessA auf)
2. Objekt ProcessA trägt sich (Referenz auf sich) in SystemScheduler (ready-queue) ein
3. Prozeß A ist jetzt im Zustand bereit und wartet darauf, daß er den Prozessor zugeteilt bekommt
4. Prozeß B blockiert sich (= Aktivitätsträger von Prozeß B ruft die Methode block an dem Objekt ProcessB auf — z. B. im Rahmen einer P-Operation)
5. Aktivitätsträger von Prozeß B ruft Methode getNextReadyProcess seines Processor-Objekts auf (Referenz auf eigenes Processor-Objekt bekommt man über die Funktion thisCPU())
6. Processor holt nächstes Prozeß-Objekt aus seinem SystemScheduler (ready-queue) (und bekommt dabei in dem Beispiel eine Referenz auf ProcessA)
7. Processor gibt Referenz auf ProcessA an ProcessB zurück
8. ProcessB ruft nun (immer noch aus der Methode block heraus) seine eigene Methode giveProcessorTo auf und übergibt Referenz auf ProcessA als Parameter
9. Stack von Prozeß A wird gesperrt (um ein gleichzeitiges Aktivieren auf einem anderen Prozessor zu verhindern!)
10. ProcessB ruft aus der Methode giveProcessorTo die architekturabhängige Methode checkpoint an dem ihm zugeordneten ProcessorContext auf (der Zustand von Prozeß B wird damit gesichert!)
11. Kontextumschaltung auf Prozeß A wird eingeleitet
12. Kontext von Prozeß A wird restauriert
13. Kontrollfluß ist jetzt in Prozeß A (der Aktivitätsträger kehrt damit ein zweites mal - diesmal mit anderem Rückgabewert - aus dem letzten checkpoint-Aufruf, den ProcessA an seinen ProcessorContext gerichtet hat, zurück!)
14. Stack von Prozeß B wird jetzt freigegeben. Der Prozeß kann damit auf einem anderen Prozessor aktiviert werden
15. Seitenkacheltabellen von Prozeß A werden installiert, falls sein Domain nicht identisch mit dem von Prozeß B ist
16. Prozeß A läuft weiter

Prozeßumschaltung (2)

❑ Code-Beispiel: Methode block() der Klasse Process

```
void Process::block()
{
    /* block suspends this process and runs next process from SystemScheduler.
     * The caller of block() should have put this process on some ProcessContainer */
    int wasInterruptable = becomeUninterruptable();
    Process * nextProcess = thisCPU()->getNextReadyProcess();
    Assert(! isInterruptable() );
    giveProcessorTo(nextProcess);
    if (wasInterruptable) {
        becomeInterruptable();
    }
}
```

Speicherverwaltung

Komponenten

❑ MemoryObject
- repräsentiert einen **Datenspeicher**, eingeteilt in Blöcke fester Länge
- Methoden: **read, write**, numberOfUnits, sizeOfUnits
- wird zur **Speicherung von Prozeßsegmenten** (Text, Daten, Stack) oder beliebigen anderen Daten verwendet
- Unterklasse PersistentStore repräsentiert einheitliche Schnittstelle für unterschiedliche Dateisysteme und Disks

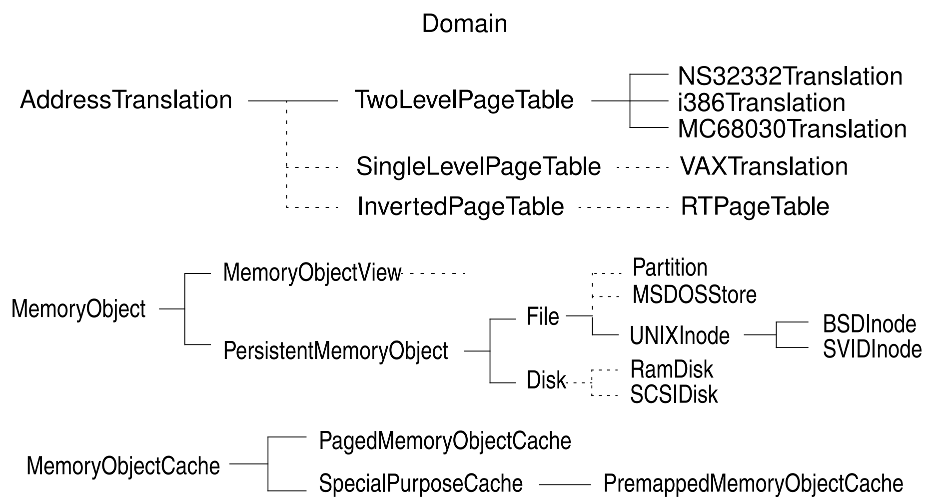
❑ Domain
- zentrale Instanz zur Verwaltung eines virtuellen Adreßraums — der Abbildung zwischen virtuellen Adressen und Datenspeichern
- Wenn ein MemoryObject in einen Domain abgebildet wird, wird ein Bereich des virtuellen Adreßraum dafür vorgesehen und ein MemoryObjectCache erzeugt, der die Daten des Datenspeichers im Hauptspeicher zwischenspeichert. Prozesse greifen auf die Daten über virtuelle Adressen zu. Liegen die Daten dabei nicht im Hauptspeicher bereit, wird von der MMU ein page-fault generiert. Die page-fault-Behandlung ruft eine Methode des zuständigen Domain-Objekts auf, das die betroffene virtuelle Adresse feststellt und eine Methode des zuständigen MemoryObjectCache-Objekts aufruft um die Einlagerung der fehlenden Daten in den Hauptspeicher zu veranlassen. Anschließend wird über das AddressTranslation-Objekt die Abbildung der virtuellen Adresse auf die Hauptspeicheradresse vorgenommen.

❑ PageFrameAllocator
- **verwaltet den Hauptspeicher**
- wird zur Reservierung und Freigabe von Kacheln für die MemoryObjectCaches verwendet

❑ AddressTranslation
- kapselt den **architekturabhängigen Teil der virtuellen Speicherverwaltung** — z. B. den Aufbau von Seitenkacheltabellen
- wird von einem Domain mit der Abbildung virtueller auf physikalische Adressen beauftragt

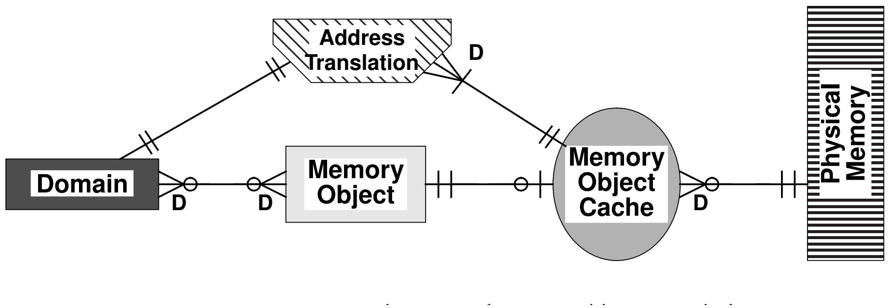
❑ MemoryObjectCache
- sorgt für die **Abbildung von Blöcken eines MemoryObjects auf Kacheln des Hauptspeichers**
- Abbildung durch den MemoryObjectCache erfolgt in **architekturunabhängiger** Form — alle architekturabhängigen Daten sind auf das AddressTranslation-Objekt beschränkt

Klassenhierarchien



- Von der Klasse Domain wurden bislang keine Unterklassen gebildet. Modifikationen von Domains erfolgen durch Ändern ihrer Komponenten — vor allem AdressTranslation und MemoryObjectCache.
- Eine Klasse vom Typ AddressTranslation kapselt die hardwareabhängige Repräsentation der Adreßabbildung von virtuellen auf physikalische Adressen. Unterklassen bilden eine Spezialisierung für verschiedene MMU-Architekturen — z. B. 2-stufige Seitenkacheltabellen, einstufige SKTs oder inverted page tables, wie sie z. B. beim IBM RT/PC eingesetzt werden. Implementiert wurden in Choices bisher allerdings nur Klassen für den NS32332, Intel386 und Motorola 68030 (die Motorola-Prozessoren unterstützen eigentlich mehrstufige (1-4) SKTs, in Choices beschränkte man sich auf die Nutzung von 2 Stufen). Die Klasse TwoLevelPageTable faßt die Gemeinsamkeiten der verschiedenen 2-stufigen SKT-Varianten zusammen.
- Die Klassenhierarchie unter MemoryObject wurde in letzter Zeit völlig neu organisiert (da die allgemeine Schnittstelle eines MemoryObjects unverändert blieb, haben diese Änderungen den Rest von Choices nicht tangiert!). Dateien und Disks bilden jetzt eine Unterklasse in PersistentMemoryObject (in der Literatur zwischenzeitlich auch als PersistentStore bezeichnet). PersistentMemoryObject ist Unterklasse von MemoryObject.
Die Gemeinsamkeiten von UNIX-Dateien werden durch die Klasse UNIXInode zusammengefaßt, die Unterschiede dann in den Klassen BSDInode und SVIDInode beschrieben.
MemoryObjectView und seine Unterklassen wird benutzt, um ein PersistentMemoryObject in einem PagedMemoryObjectCache zu halten und so für die Abbildung in einen Domain geeignet bereitzustellen.
- Im allgemeinen wird die Klasse PagedMemoryObjectCache als MemoryObjectCache eingesetzt. Diese Art von Cache sorgt für eine Ein- und Auslagerung von MemoryObject-Teilen auf der Basis von Kacheln. PremappedMemoryObjectCache ist eine andere Cache-Implementierung, die große, zusammenhängende Speicherbereiche für die Zwischenspeicherung von MemoryObjects verwendet und die Daten sofort komplett in den Hauptspeicher einlagert.

Klassenbeziehungen



1. Domain — MemoryObject
   - einem Domain sind (optional) ein oder mehrere MemoryObjects zugeordnet
   - MemoryObjects können in mehrere Domains abgebildet sein
   - die Zuordnung von MemoryObjects zu Domains kann während der Laufzeit verändert werden

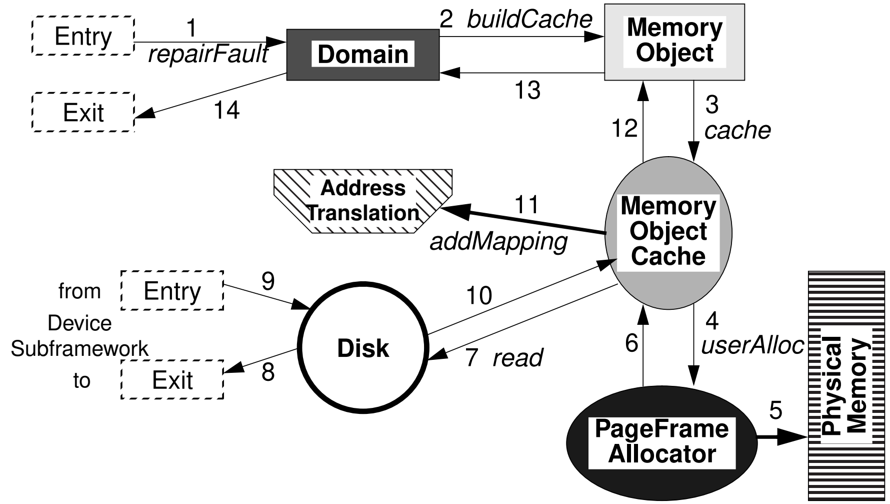
2. MemoryObject — MemoryObjectCache
   - jedem MemoryObject, das in einen Domain abgebildet ist, ist ein MemoryObjectCache zugeordnet, der die Adreßabbildung in architekturunabhängiger Form vornimmt

3. MemoryObjectCache — PhysicalMemory
   - der Hauptspeicher wird an verschiedene MemoryObjectCaches zur Zwischenspeicherung von MemoryObject-Teilen vergeben

4. Domain — AddressTranslation
   - jedem Domain ist genau ein AddressTranslation-Objekt zugeordnet, daß die architekturabhängigen Datenstrukturen für den virtuellen Adreßraum des Domains verwaltet

5. MemoryObjectCache — AddressTranslation
   - für jede Hauptspeichereinheit (Kachel) führt ein MemoryObjectCache Buch über die AdressTranslation-Objekte, die diese Kachel in ihrem virt. Adreßraum abgebildet haben (um die Abbildung bei ein *page-out* zu invalidieren oder um Statistik-Informationen anzufordern). Eine Kachel kann in mehrere Domains abgebildet und damit in mehreren AddressTranslation-Objekten eingetragen sein.

Kontrollfluß (Beispiel page-fault)



1. Die Interrupt-Bearbeitung (page-fault) ruft die Methode repairFault des zuständigen (gerade auf der CPU aktiven) Domains auf.
2. Der Domain ermittelt das zu der betroffenen Adresse gehörende MemoryObject und ruft dort dieMethode buildCache auf. Dadurch wird, wenn nötig, ein neuer MemoryObjectCache instantiiert.
3. Das MemoryObject ruft die cache-Methode seines memoryObjectCaches auf und übergibt eine Referenz auf das AddressTranslation-Objekt des Domains.
4. Der MemoryObjectCache besorgt sich über die Methode userAlloc eine Kachel von der zentralen Hauptspeicherverwaltung (PageFrameAllocator)
5. Der PageFrameAllocator sucht freien Platz im Hauptspeicher und stößt bei Bedarf die Auslagerung von Seiten an (hier nicht weiter beschrieben).
6. Der PageFrameAllocator gibt eine Referenz auf ein freies PAU-Objekt (PhysicallyAdressableUnit) an den MemoryObjectCache zurück.
7. Der MemoryObjectCache fordert die Daten der Seite von einem Disk-Objekt (oder auch einem anderen MemoryObject-Objekt) an.
8. Der Kontrollfluß verläßt die virtuelle Speicherverwaltung, die Einlagerung wird von dem Disk-Subsystem vorgenommen.
9. Nach erfolgreicher Einlagerung kehrt der Kontrollfluß aus dem Disk-Subsystem zurück.
10. Die Seite wird dem MemoryObjectCache übergeben.
11. Der MemoryObjectCache sorgt für die Eintragung der Adreßabbildung in den hardware-abhängigen Datenstrukturen (AddressTranslation-Objekt).
12. Der Aufruf der cache-Methode kehrt zurück
13. Der Aufruf der buildCache-Methode kehrt zurück.
14. Der unterbrochene Prozeß wird fortgesetzt.

MemoryObjects

Unterklassen

- ❑ Disk
  - – entspricht einem abstrakten Disk-Treiber
    - • Unterklassen realisieren Implementierungen konkreter Disk-Treiber, wie z. B. SCSI-Disk, RAM-Disk, PS2Diskette
- ❑ File
  - – steht für eine virtuelle Speichereinheit
  - – jedem File-Objekt ist ein Quell-PersistentMemoryObject zugeordent
  - – präsentiert spezielle Sicht auf dieses andere PersistentMemoryObject
  - • File stellt damit eine Art Fenster auf ein PersistentMemoryObject dar
    - – Größe des Fensters kann fest oder variabel sein
    - – es kann ein Teil oder das ganze Quell-PersistentMemoryObject repräsentiert werden
    - – das Fenster kann ein zusammenhängendes Stück oder mehrere Teile eines Quell-PersistentMemoryObject darstellen
  - • Am Ende resultiert ein read oder write auf ein File-Objekt aber immer in einem read oder write auf ein Disk-Objekt
  - • Beispiele:
    - – Aufteilung einer Disk in Partitionen, Partitionen in Dateien, etc.
    - – UNIXInode stellt Datei auf einer Partition dar
    - – TarStore stellt Inhalt eines UNIXInodes dar
  - • Ein File-Objekt muß das PersistentMemoryObject, in dem es gespeichert ist kennen und wissen, wie es die Informationen dort zu interpretieren hat. Beispiel:
    - – Ein BSDInode-Objekt muß wissen, wie die Block-Nummer des BSDInode in die entsprechende Blocknummer der darunterliegenden Partition umgerechnet wird.
    - – Ein Partition-Objekt muß eine Blocknummer in die Diskdaten, die ein Disk-Objekt benötigt umwandeln.

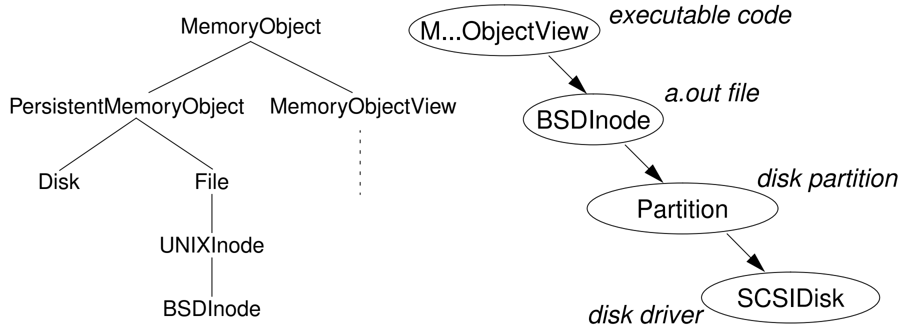
Aufbau neuer MemoryObjects

- • MemoryObjects sind
  - – direkte Repräsentationen von Disks oder physischem Speicher
  - – Ensembles anderer, einfacherer MemoryObjects
- ❑ Anpassung & Erweiterung der Choices-Speicherverwaltung
  - • Kombination existierender Versionen von MemoryObjects zum Aufbau neuer Ensembles
- ❑ Schnittstellen des Frameworks garantieren Kompatibilität
  - • Vererbung vereinfacht Erzeugung neuer MemoryObject-Klassen
  - • Domain akzeptiert jedes MemoryObject als Repräsentant für einen Bereich des virt. Adreßraums

MemoryObjects (2)

Vererbungsbeziehung ↔ Aggregationsbeziehung

- ❑ wichtiger Unterschied zwischen
  - – Wiederverwendung im Rahmen von Vererbung und
  - – Wiederverwendung durch Kombination von Objekten



- • Im linken Teil der Abbildung wird eine normale Klassenhierarchie durch Vererbung beschrieben. Die konkreten Subklassen erben jeweils die Implementierung und Schnittstellen (und damit vor allem auch den Typ) ihrer Superklassen.
- • Der rechte Teil der Abbildung zeigt dagegen, wie komplexe Objekte durch Aggregation anderer, existierender Objekte aufgebaut werden können. Es wird ein Ensemble dargestellt, das das Code-Segment eines Prozesses darstellt. Letztlich ist der Programm-Code auf einer konkreten Disk — in dem Beispiel einer SCSIDisk — abgelegt. Die Disk wird durch MemoryObjectViews in Partitionen unterteilt, der Programm-Code liegt einer solchen Partition. Die Partition ist in Dateien eines BSD-Dateisystems aufgeteilt, die Datei, in der der Programm-Code liegt wird durch ein File-Objekt, nämlich einen BSDInode verwaltet.
- • Das Laden eines Programms in einen neuen Domain geht dann folgendermaßen vor sich:
  - – Beginn und Länge von Text- und Datensegment werden ermittelt (aus der a.out-Datei) und es wird dafür jeweils ein NormalMemoryObjectView (Unterklasse von MemoryObjectView erzeugt. Diser wird in den neu zu erzeugenden Domain abgebildet.
  - – Ein MemoryObjectView-Objekt erzeugt einen ObjectCache (PagedMemoryObjectCache) für seine Daten.

System-Schnittstelle

- ❑ Ziel: Systemaufrufe = Methodenaufrufe an Objekte des Betriebssystems
- ❑ Probleme:
  - Aufrufer im Benutzer-Adreßraum, Ziel im Kernel
  - Referenz (Zeiger) auf BS-Objekt
  - Parameterübergabe
- ❑ Konventionelle Lösung: spezielle Funktionen & SVC/trap-Befehl
- ❑ Lösung in Choices:
  - Proxy-Objekte: Stellvertreter für Systemobjekte
  - Nameserver: Systemaufruf-Name → Proxy-Referenz
    - Anwendungsobjekte erhalten Referenzen auf Proxy-Objekte mit Hilfe der lookup-Methode des NameServers. Das Anwendungsobjekt benennt das gewünschte Systemobjekte dabei über einen symbolischen Namen.
    - Die NameServer-Schnittstelle wird durch eine abstrakte Klasse NameServer festgelegt - es sind verschiedene konkrete Unterklassen denkbar, die die Abbildung unterschiedlich vornehmen.
    - Jeder Anwendung wird initial ein *default*-NameServer zugeordnet.
    - Das Betriebssystem kann mit Hilfe der bind-Methode neue Systemobjekte bei einem Nameserver registrieren.
    - Nur die speziell als *proxiable* gekennzeichnete Schnittstellen registrierter Systemobjekte sind aus Anwendungen heraus aufrufbar.
    - Dem Benutzer werden *Header-Files* mit den *proxiable*-Schnittstellen der aufrufbaren Systemobjekten zur Verfügung gestellt.
    - Beim Aufruf von Methoden von Systemobjekten werden Sicherheitsprüfungen durchgeführt.

Proxy-Objekte

- ❑ allgemein: lokaler Repräsentant für ein entferntes Objekt
  - transparenter Zugriff auf Systemobjekte
  - transparenter Zugriff auf Objekte in anderen Adreßräumen
  - transparenter Zugriff auf Objekte auf anderen Knoten eines verteilten Systems

Proxy-Objekte als Systemaufruf-Schnittstelle

- ❑ abgebildet in *read-only*-Adreßbereich des Benutzeradreßraums
  - ➔ Zustand des Proxy-Objekts vom Anwendungsprogramm nicht modifizierbar
  - ➔ Proxy-Objekte zwischen verschiedenen Domains *sharable*
- ❑ lösen über trap Einsprung in Systemkern aus
  - Anwendungszustand wird dabei gesichert
  - Methodennummer, Aufrufparameter und der *this*-Zeiger werden für Kern abgelegt
  - Systemkern ermittelt aufzurufendes Systemobjekt aus gesichertem Anwendungszustand
- ❑ Referenzen (Zeiger) auf Proxy-Objekte an Systemobjekte übergebbar

  Proxy-Objekte auch im Systemkern gültige Stellvertreter für Systemobjekte
  - von Parametern, die für Proxy-Objekte stehen, wird nach Einsprung in den Kern die Proxy-Information entfernt:
    - ➔ entweder bleibt eine Referenz auf ein Kern-Objekt

    oder das Zielobjekt liegt in einem Anwendungs-Domain
    - ➔ Kern erzeugt Proxy hierfür, so daß bei einem Aufruf in den entsprechenden Domain geschaltet wird
- ❑ neue Systemobjekte instantiierbar
  - Falls die Konstruktor-Methode einer Systemklasse als *proxiable* deklariert ist, können Objekte dieser Klasse aus einer Anwendung heraus instantiiert werden. Das Objekt selbst entsteht dabei allerdings im Systemkern, in der Anwendung entsteht lediglich ein Proxy-Objekt. Bei einem delete wird das ProxyObjekt gelöscht, das dahinter stehende Systemobjekt nur dann, wenn es nicht mehr anderweitig referenziert wird.
- ❑ Subklassen von Systemobjekten
  - Innerhalb einer Anwendung können Subklassen von Systemklassen gebildet werden. In solch einer Subklasse werden *proxiable*-Methoden der Basisklasse geerbt. Es können wie bei normaler Vererbung weitere Methoden definiert werden und auch Methoden der Basisklasse redefiniert werden. Wird bei einer Instanz solch einer Subklasse eine neu- oder redefinierte Methode aufgerufen, wird die Methode ganz normal innerhalb des Anwendungs-Domains ausgeführt. Wird eine Methode aufgerufen, die von der Basisklasse geerbt wurde, erfolgt ein Wechsel in den System-Domain und die Methode wird dort ausgeführt. Neu- oder redfinierte Methoden haben allerdings (natürlich) keinen Zugriff auf *protected*- oder *public*-Daten der Basisklasse. Ruft eine Basisklassen-Methode eine, in der Subklasse redefinierte Methode auf, so erreicht sie (obwohl diese als *virtual* deklariert war) nicht die Implementierung der Subklasse im Anwendungs-Domains sondern die Implementierung der Basisklasse im System-Domain. Es handelt sich damit nicht um richtige Vererbung über Domaingrenzen hinweg.

Beispiel

```
#include "FileSystem.h"
#include "FileSystemInterface.h"
#include "FileSystemInterfaceStar.h"
#include "NameServer.h"
#include "NameServerStar.h"
#include "OutputStream.h"
#include "OutputStreamStar.h"
#include "PersistentArrayStream.h"
#include "PersistentArrayStreamStar.h"

#define BUFSIZE 80

int main(int argc, char *argv[])
{
  OutputStreamStar stdout = StandardOutput; /* als Abkuerzung fuer StandardOutput */

  int i,status,more_than_one,no_of_bytes;
  char buffer[BUFSIZE];

  if(argc<2)
    {
      stdout->formattedWrite("usage: cat filename [filename..]\n");
      return(-1);
    }

  /* falls mehr als ein File angegeben wurde: Filenamen mit ausgeben */
  more_than_one = (argc>2);

  /* ObjectProxy fuer das FileSystemInterface holen, durch Anfrage beim NameServer */
  FileSystemInterfaceStar FSI =
    StandardNameServer->lookup("fileSystemInterface","FileSystemInterface");
  Assert(FSI != 0)("FileSystemInterface not found\n");

  /* Klassenobjekt(ObjectProxy) fuer die Klasse PersistentArrayStream holen */
  ClassStar PersistentArrayStreamClass =
    StandardNameServer->lookup("PersistentArrayStream","Class");
  Assert(PersistentArrayStreamClass)("PersistentArrayStreamClass not found\n");

  for(i=1;i<argc;i++)
    {
      /* Datei argv[i] als Objekt der Klasse PersistentArrayStreamClass oeffnen*/
      PersistentArrayStreamStar file =
         FSI->open(argv[i],PersistentArrayStreamClass,status,O_RDONLY);
      if (file != 0)
         {
           if(more_than_one)
             stdout->formattedWrite(">>>%s\n",argv[i]);
           no_of_bytes = file->read(buffer,BUFSIZE-1,status);
           while(status == 0)
             {
               buffer[no_of_bytes]='\0';
               stdout->formattedWrite("%s",buffer);
               no_of_bytes = file->read(buffer,BUFSIZE-1,status);
             }
         }
      else
         stdout->formattedWrite("couldn't open %s : %s \n",
                                        argv[i],StatusCodes[status]);
    }
  return(0);
}
```


Beispiel

```
#include "FileSystemInterface.h"
#include "FileSystemInterfaceStar.h"
#include "NameServer.h"
#include "NameServerStar.h"
#include "PersistentArrayStream.h"
#include "PersistentArrayStreamStar.h"

int main(int argc, char *argv[])
{
...
  /* ObjectProxy für FileSystemInterface von NameServer besorgen */
  FileSystemInterfaceStar FSI = StandardNameServer->lookup
                    ("fileSystemInterface","FileSystemInterface");
...
  /* Klassenobj.(ObjectProxy) für Klasse PersistentArrayStream holen */
  ClassStar PersistentArrayStreamClass =
    StandardNameServer->lookup("PersistentArrayStream","Class");
...
/* Datei argv[i] als Obj. der Klasse PersistentArrayStreamClass öffnen */
  PersistentArrayStreamStar file = FSI->open
            (argv[i],PersistentArrayStreamClass,status,O_RDONLY);
...
  no_of_bytes = file->read(buffer,BUFSIZE-1,status);
...
```

Erweiterungen für C++

- ❑ Garbage collection
  - – Mechanismen zur automatischen Vernichtung nicht mehr referenzierter Objekte
    - • Normalerweise werden C++-Objekte explizit erzeugt und vernichtet (new / delete).
    - • Wenn Objekte, die Ressourcen repräsentieren, in mehreren Domains (Prozessen) gemeinsam genutzt werden, wird es für den Programmierer schwierig, zu entscheiden, wann ein Objekt gelöscht werden kann.
    - ➔ Referenzzähler und Zeiger-Objekte
      - – alle Objekte, die automatisch gelöscht werden sollen, erben von einer Klasse Object einen Referenzzähler und die Methoden reference(), unreference(), noRemaingReference(), Object() und ~Object().
      - – ObjectStar-Klasse für Zeiger auf Objekte mit Referenzzählern mit Konstruktor, Destruktor und Zuweisungsmethode, die die reference()- und unreference()-Methoden der Zielobjekte entsprechend aufrufen.

- ❑ Repräsentation von Klassen als Objekte zur Laufzeit
  - • Typprüfungen in C++ nur statisch → Probleme bei der Übergabe von Objektreferenzen zwischen Domains (Typprüfung nur zur Laufzeit möglich). Klassen, die als Objekte zur Laufzeit existieren, ermöglichen solche Typprüfungen.
  - • Debugging (z. B. Anzeige von existierenden Klassenhierarchien zur Laufzeit)
  - • persistente Objekte — Sichern von Typinformation
  - • bessere Kontrolle bei dynamischer Erweiterung des Systems — z. B. Nachladen neuer Subklassen
  - • Realisierung: Klassen = Instanzen einer Klasse Class

- ❑ dynamisches Laden von neuen Unterklassen und Code