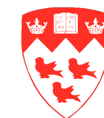
## Software Fault Tolerance

# Implementing Transactions

Jörg Kienzle
Software Engineering Laboratory
School of Computer Science
McGill University

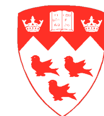


McGill

# Overview

(Kienzle chapters 4-8)

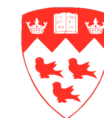
- Transaction Context
- Concurrency Control
- Persistence
- Caching
- Crash Recovery
- Interfaces for the Programmer
- Customization



McGill

# Transaction Library Requirements

- Handle transaction life-cycle
  - Starting, aborting, committing
  - Monitor nesting relationship
- Monitor accesses to transactional objects
  - Apply concurrency control
  - Handle object updates, checkpointing and undoing
  - Gather recovery information in case of crash failures
- In case of a crash, perform recovery after restart
- Provide elegant interface to the programmer
  - Easy to use
  - Safe to use



McGill

# OPTIMA [K+01]

- Object-oriented framework for transactional systems
  - Defines architecture, classes, objects and their responsibilities
  - Programming language independent
- Customizable and extensible
  - Strict | semantic-based concurrency control
  - Pessimistic | optimistic concurrency control
  - Various recovery and caching strategies
  - In-place | deferred update for objects
  - Physical | logical logging
  - Extensible storage support
- Based on design patterns [GHJV95]

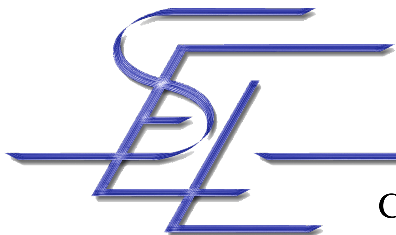
# Optima Architecture

Application Code

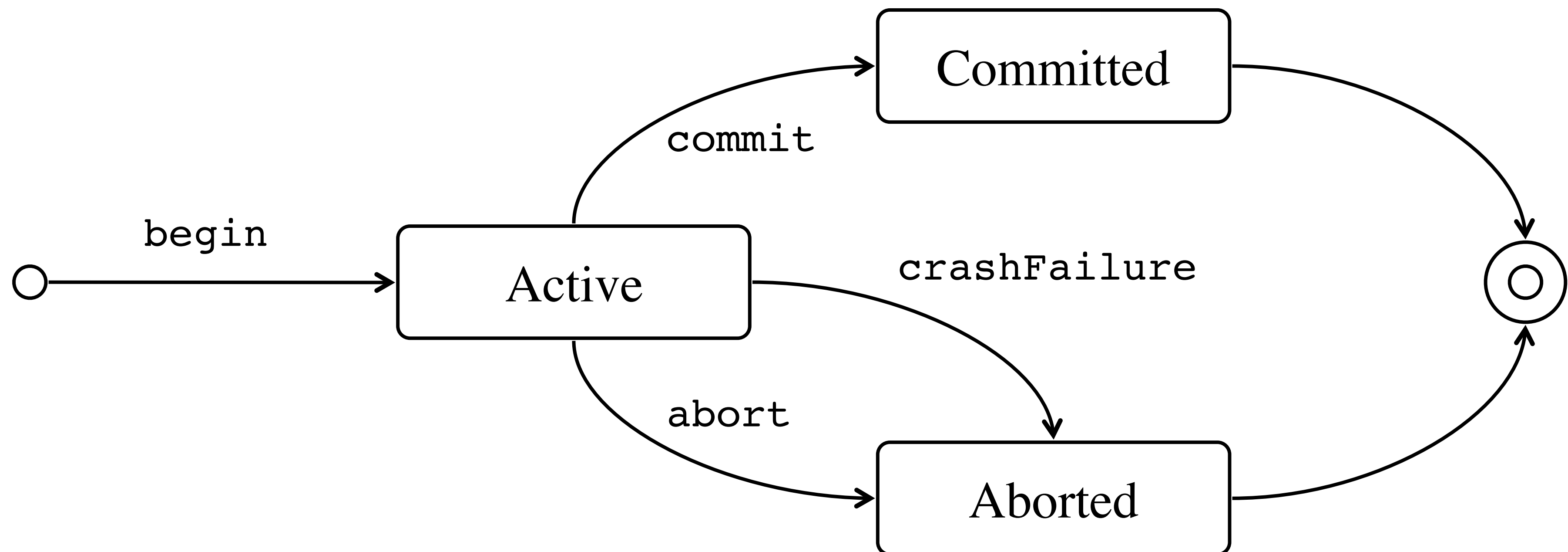
Interface

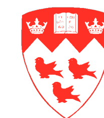
Transaction Life-Cycle Management

| Concurrency Control | Memory Management | Recovery Support |
|---|---|---|



# Transaction Life-Cycle Management





McGill

# Transaction Context

- During the life-time of a transaction, its current state, the unique transaction identifier, and the set of accessed transactional objects are stored in the transaction context
  - All transactional objects must be notified in case of abort or commit
- The context also keeps track of the parent and child transactions
- In case of commit, the responsibility of all operation invocations of a child transaction must be handed over to the parent transaction

# Transaction Class

Transaction

TID myID
Status myStatus

beginTransaction()
commitTransaction()
abortTransaction()
addTransactionalObject()
addChildTransaction()

parent 1

0..* children

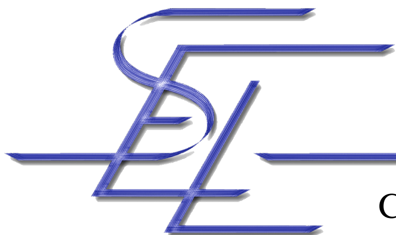
Subtransaction

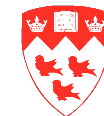
0..* accessedObjects

TransactionalObject

<<enum>>
Status

active
committed
aborted

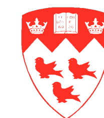




McGill

# Begin Transaction

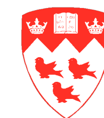
- Create a new transaction context
  - If begin is called from within a transaction, create a subtransaction
- Initialize the context
  - Set the transaction status to active
  - Obtain a new transaction identifier
  - For nested transactions, setup the parent-child relationship



McGill

# Abort Transaction

- Set transaction status to aborted
- Inform recovery support of abort
  - Record the abortion in the log
- For every accessed transactional object
  - Undo the changes made by the transaction (if needed)
  - Notify the associated concurrency control of the abort
  - For nested transactions
    - Remove oneself from the parents child list



McGill

# Commit Transaction

- Ask the concurrency control of every transactional object to validate the transaction
  - Required for optimistic concurrency control
- If successful and it's a top-level transaction, then perform 2-phase commit
  - Ask every accessed transactional object to prepare for commit
  - Set transaction status to committed
  - Record the commit in the log ← Point of "no return"
- For every accessed transactional object
  - Notify the associated concurrency control of the commit
- For nested transactions
  - Add the accessed objects to the accessed object list of the parent
  - Remove oneself from the parents child list

# Concurrency Control

- Concurrency control must be applied for every transactional object
  - Before and after every operation invocation
  - At validation time (for optimistic approaches)
  - Upon transaction abort and commit
- The optimal concurrency control strategy is application dependent
  - Maybe even object-dependent
- Customization based on the *Strategy* design pattern
  - Abstract `ConcurrencyControl` class
  - The user can choose the appropriate concrete concurrency control for each object
  - The user can extend the hierarchy and implement his own concurrency control

# Strategy Design Pattern

Context

1

currentStrategy

*Strategy*

*interface()*

Contains code that makes calls to the methods declared in the *Strategy* interface

Concrete Strategy1

interface()

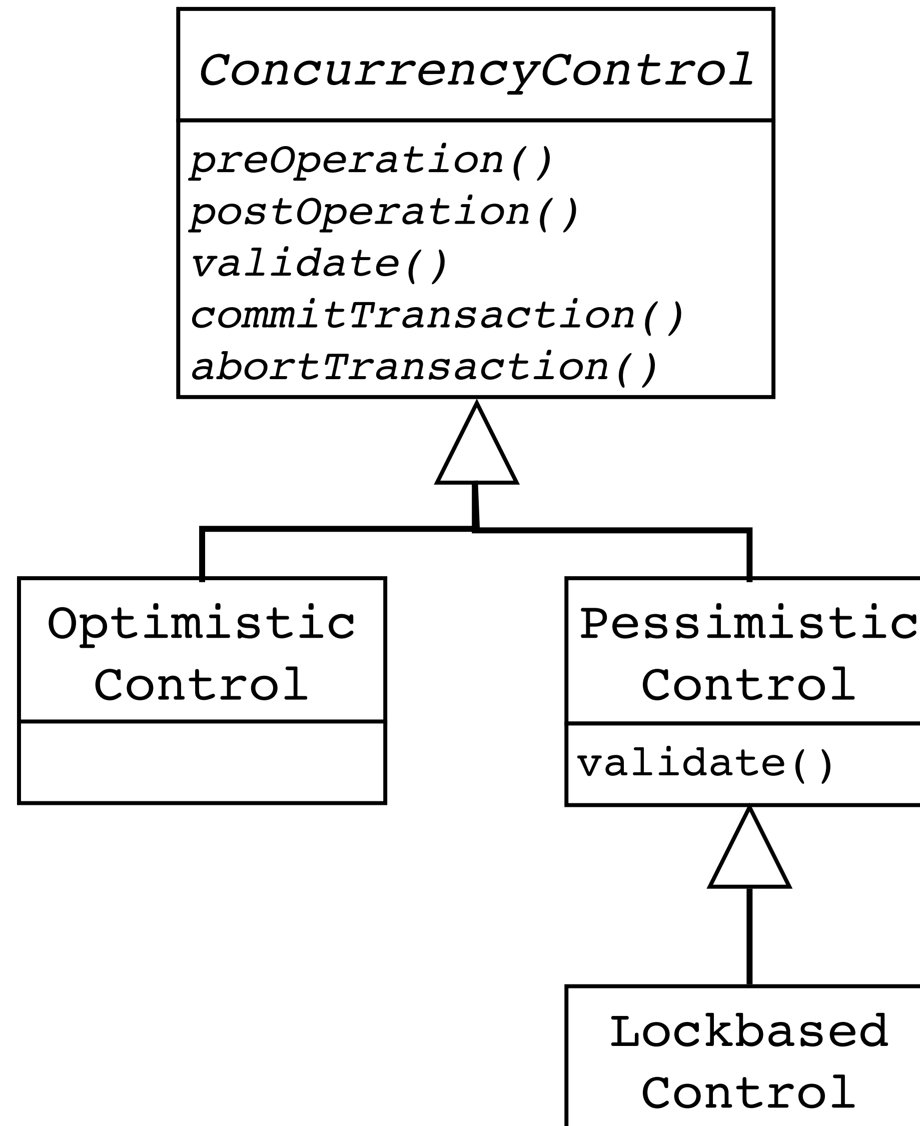
...

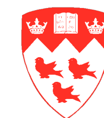
Concrete StrategyN

interface()

McGill

# Concurrency Control Hierarchy





McGill

# Lock-based Concurrency Control

- Conceptually, every operation on a transactional object has an associated lock
- A transaction wanting to perform an operation on a transactional object must acquire the corresponding lock first
- If incompatible locks are held by other active transactions, the invoking thread is blocked
- Blocked requests are served as soon as possible in FIFO order
- The lock-based concurrency control keeps two list
  - Granted locks (to active transactions)
  - Requested locks (by blocked threads)
- Implementation must be thread-safe

# PreOperation & PostOperation

- PreOperation
  - Go through the list of granted locks, checking for conflict
    - Remember, locks held by the parent transaction do not conflict. They can be claimed by the child!
  - If there is a conflict
    - Block the invoking thread
    - Insert the lock request into the waiting requests list
  - If there is no conflict
    - Insert the lock into the list of granted locks
- PostOperation
  - Do nothing!

# AbortTransaction

- Go through the list of granted locks
  - If the lock belongs to the aborting transaction
    - If it has been claimed from the parent, pass it back to the parent
  - Else delete it
- Go through the list of lock requests and check if any request can now be granted
  - If yes, remove the request from the list, insert the corresponding lock into the granted locks list, and awake the suspended thread

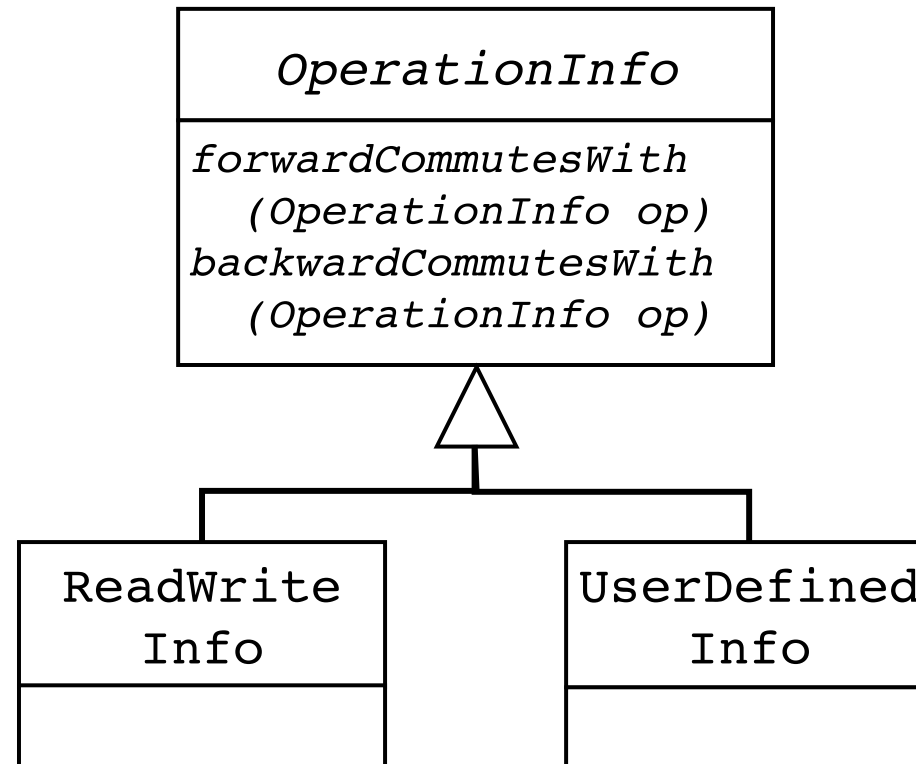
# CommitTransaction

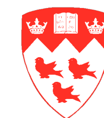
- Go through the list of granted locks
  - If the lock belongs to the committing transaction
    - If the committing transaction is a top-level one
      - Delete the lock
    - Else (it has either been claimed from the parent transaction, or it is a new lock acquired by the child transaction)
      - Change the owner of the lock (back) to the parent transaction
- Go through the list of lock requests and check if any request can now be granted
  - If yes, remove the request from the list, insert the corresponding lock into the granted locks list, and awake the suspended thread

# Semantic-based Concurrency Control

- If we have no knowledge of the semantics of an operation, we must assume that all operations conflict
- Idea: make it possible for the programmer to pass semantic information in form of a commutativity table to the transaction support
- The commutativity information is passed to the concurrency control upon every operation invocation

# Operation Information Hierarchy





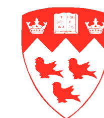
# The `ReadWriteInfo` Class

```
public class ReadWriteInfo extends OperationInfo {

  private boolean ReadWrite;
  // Read = false, Write = true

  public ReadWriteInfo(boolean rw) {
    ReadWrite = rw;
  }

 public boolean backwardCommutesWith(OperationInfo op) {
    return (this.ReadWrite == false) &&
      (((ReadWriteInfo)op).ReadWrite == false);
  }
  public boolean forwardCommutesWith(OperationInfo op) {
    return
     (this.ReadWrite == true) ||
     ((ReadWriteInfo)op).ReadWrite == false);
  }
}
```



McGill

# Persistence

- The state of transactional objects must be stored on stable storage [LS79] in order to be durable
- The transaction history is stored in a log on stable storage
- How can we build stable storage?
  - Use redundancy
  - Techniques depend on failure assumptions

# Classification of Storage [KR02]

basic data saving and restoration operations

*Storage*
*read(..)*
*write(..)*

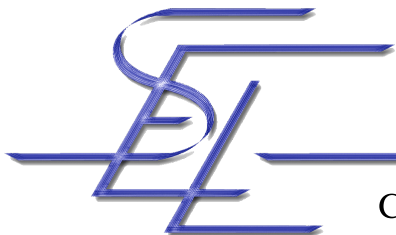
Volatile Storage

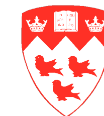
NonVolatile Storage
open(..)
close(..)
delete()

housekeeping operations

Stable Storage

NonStable Storage

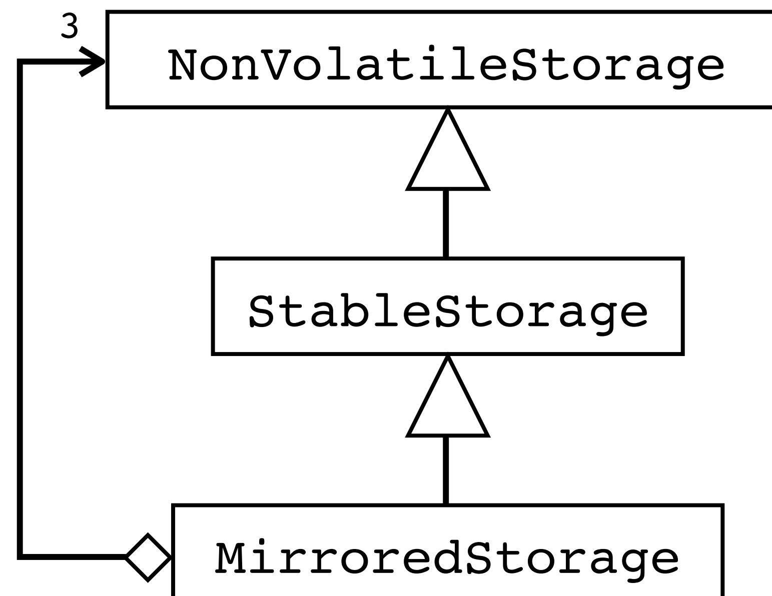




McGill

# Stable Storage based on Mirroring [CKS01]

- Idea
  - Keep two copies of the data (A and B), update sequentially
  - Write the second copy only if the first write was successful
  - Remember the current state in a log



McGill

# Failure Assumptions

- Non-volatile storage: Copy A, B and the log are stored on non-volatile storage, i.e. their data survives program termination
- Non-destructive reads: Reading from storage will not corrupt the data
- Failure isolation: A crash that occurs during a write operation on a storage unit can only corrupt the storage unit written to
- Unbuffered writes (or the availability of a flush operation that forces internal buffers to be emptied)

# Mirrored Read / Write Operation

- Write
  - Set log to *writing A*
  - Write data to A
  - Set log to *writing B*
  - Write data to B
  - Set log to *idle*
- Read
  - Read A or B

# Restart after Crash

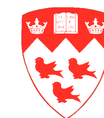
- Perform cleanup based on the following table

| State of the Log | Suspected Problem | Cleanup Action |
|---|---|---|
| *Idle* | No Problem | None |
| *Writing A* | A was not successfully written and might be | Copy B to A<br>Set log to *Idle* |
| *Writing B* | B was not successfully written and might be | Copy A to B<br>Set log to *Idle* |
| *Corrupted* | Neither A nor B is corrupted, but they might contain different data | Copy A to B<br>Set log to *Idle* |

McGill

# Identifying Transactional Objects

- There must be a unique way to identify a transactional object that survives program termination
- Based on the identifier, it should be possible to “find” the associated state on the stable storage
- Idea
  - Storage parameter hierarchy with the same structure as the storage hierarchy
  - Every storage parameter must be mappable to a string representation
  - Each storage parameter must be capable of creating and initializing the associated transactional object
  - Storage parameters must be serializable


McGill

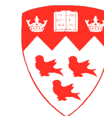
# Storage Parameter Hierarchy

```
StorageParameter
  stringToStorageParams(..)
  storageParamsToString(..)
  Storage create(..)

VolatileParameter        NonVolatileParameter
                           NVStorage open()

StableParameter          NonStableParameter
```

Factory methods



McGill

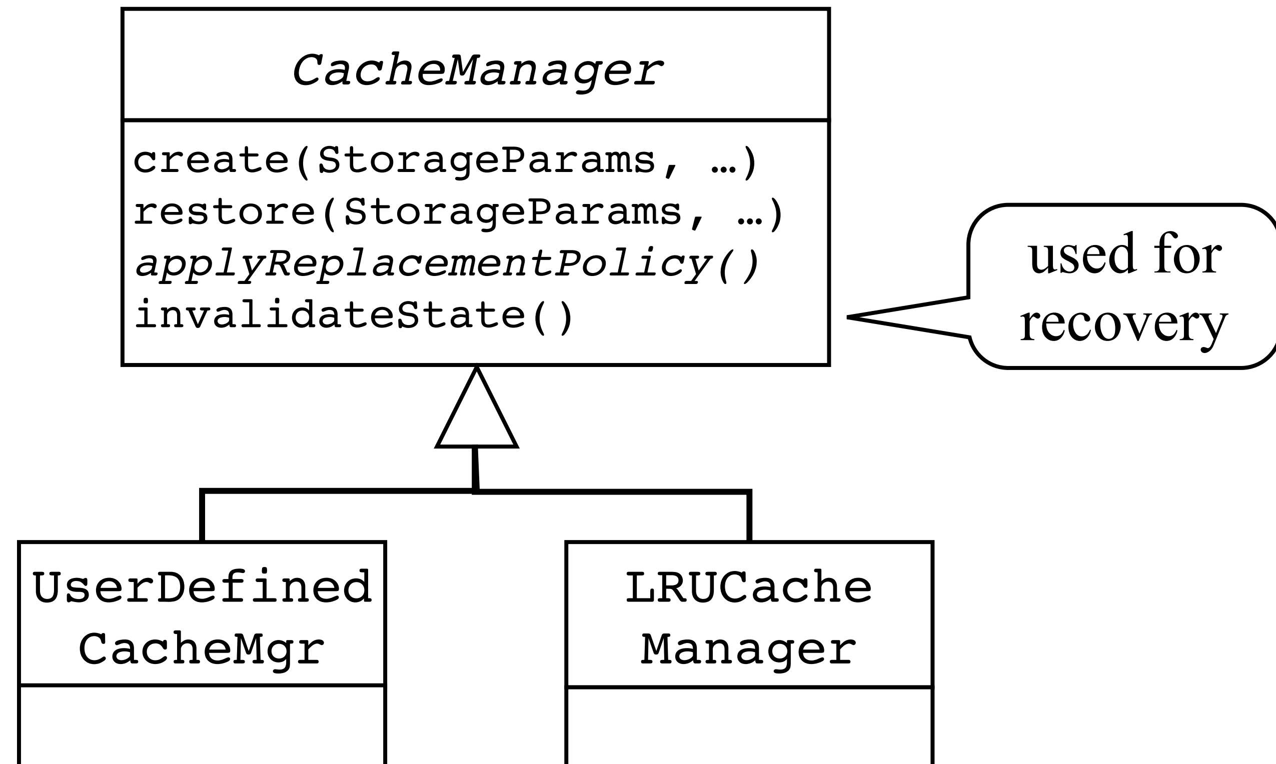
# Caching

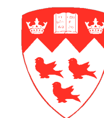
- Sometimes it is not possible to keep the state of all transactional objects in memory
- Idea
  - Use a cache
- Different cache policies
  - Load objects on demand
  - Prefetch objects by “guessing” object that are likely to be accessed in the future
- Optimal strategy is application dependent

# Cache Replacement Policies

- Least-Recently-Used
  - Monitor accesses of transactional objects
  - Select the least recently used one for replacement
- In transactional systems, not just any object can be replaced, it depends on the recovery strategy
  - No-Steal: Objects modified by a transaction in progress can not be replaced
  - Force: Objects modified by a transaction are forced to stable storage on transaction commit

# Cache Manager Hierarchy




McGill

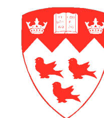
# Memory Objects

- Instances of the class `MemoryObject` handle loading and saving the state of transactional objects
- Pinned objects can not be propagated to storage

| MemoryObject |
| --- |
| load(StorageParameter p)<br>save(StorageParameter p)<br>pin()<br>unpin()<br>... |

# Consequences of using a Cache

- Transparent for the user
- Complicates things for recovery
- At any given time, the current state of a transactional object is determined first by the state in the cache, and, if it's not in memory, by the state stored in stable storage
- The contents of the cache are lost when the system crashes, therefore
  - The storage might not contain updates of committed transactions — Redo
  - The storage might contain updates of uncommitted transactions (that will abort due to the crash) — Undo



McGill

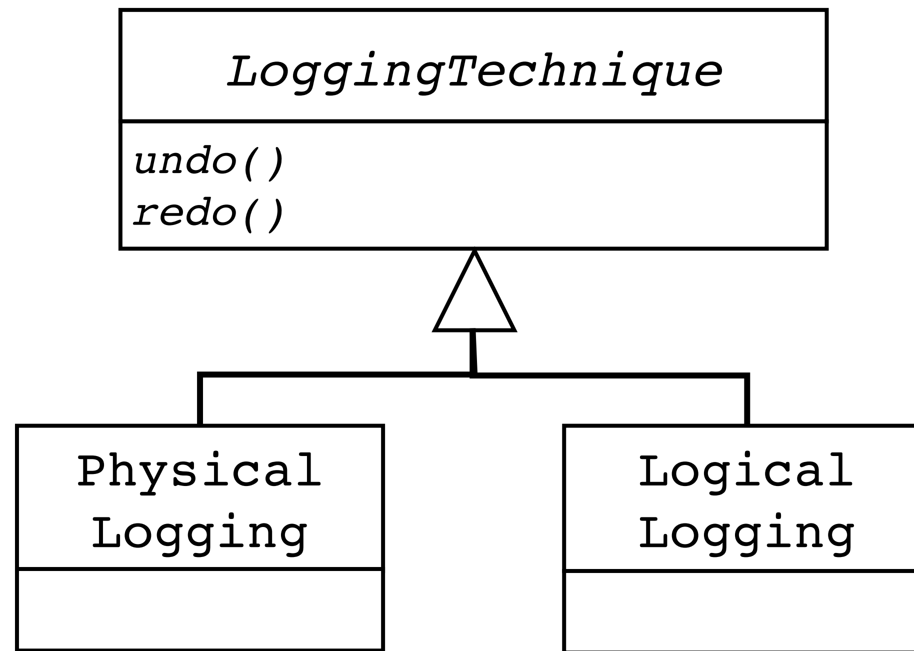
# Log

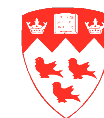
- Sequential file stored on stable storage
- Contains data needed for crash recovery
  - Transaction status information
  - Transactional object creation information
  - Transactional object deletion information
  - Operation undo information
  - Operation redo information
- The log is updated when
  - A transaction commits or aborts
  - A transactional object is created or destroyed
  - An operation modifying the state of a transactional object is invoked

# Logging Techniques

- Physical logging
  - Store before- or after-checkpoints of the state of transactional objects
  - Works only with strict concurrency control
- Logical logging
  - Log operation invocation together with all parameters
  - In order to support undo, every operation must have an associated inverse operation that undoes its effects

# Logging Technique Hierarchy





# Log Information Hierarchy

*LogInformation*
TID t

TransactionInformation
Status s

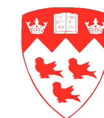
ObjectInformation
StorageParameters o

CreationInfo

UndoInfo
LoggingTechnique l

RedoInfo
LoggingTechnique l

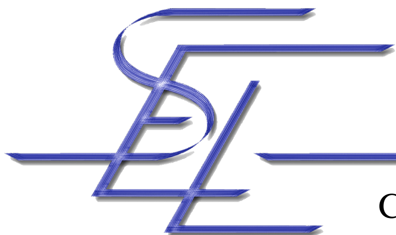
DeletionInfo
PhysicalLogging l

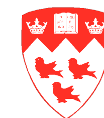


McGill

# Caching and Recovery

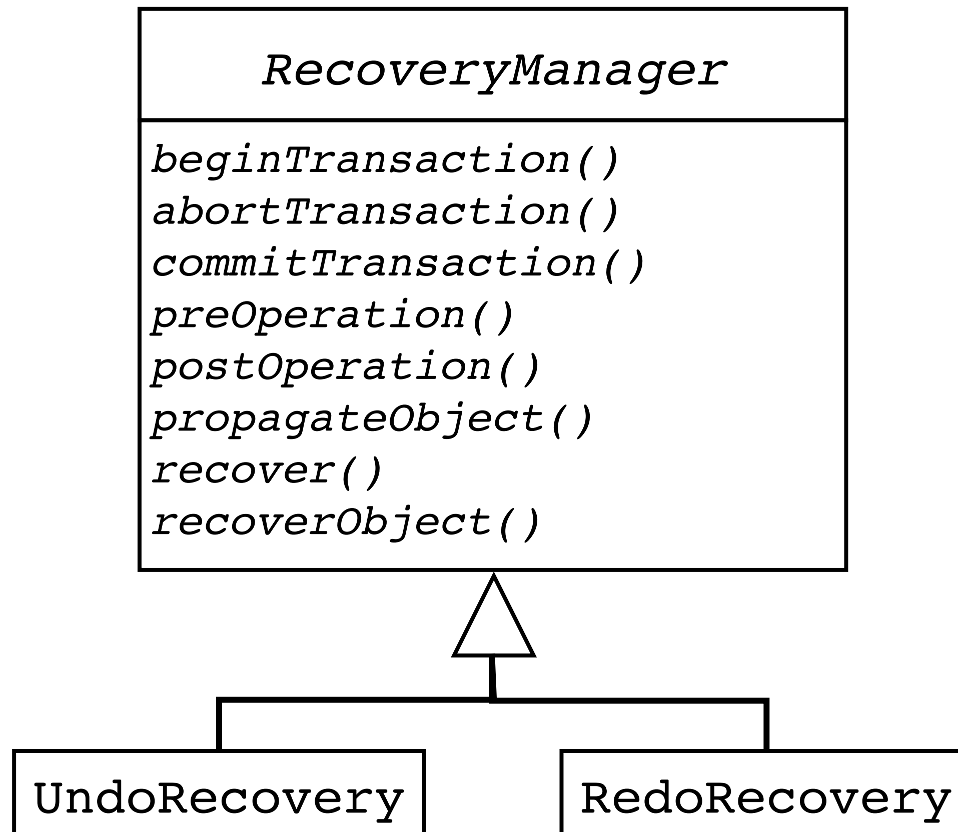
- Trade-off between flexible caching and amount of work to be done when preforming recovery after a crash
- Undo/Redo allows Steal / NoForce policies
- Undo/NoRedo makes sure that all changes are written to stable storage before transaction commit (Steal combined with Force policy)
- NoUndo/Redo keeps after-images or intention lists of operations in order to redo committed changes after a crash (NoSteal combined with NoForce)

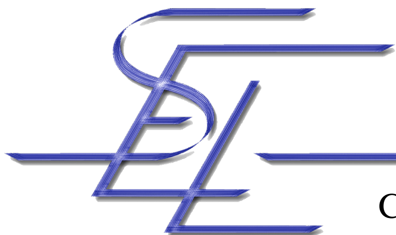


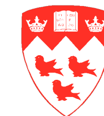


McGill

# Cache Manager Hierarchy



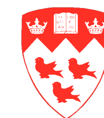





McGill

# Recovery Manager Operations

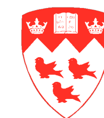
- `begin`, `abort` and `commit` called when the transaction state changes
- `pre`- and `postOperation` called before and after every operation invocation on a transactional object
- `propagate` is called by the cache manager before it replaces a transactional object's state
- `recover` is called upon restart in case of a crash failure
  - Tags potential inconsistent objects by calling `invalidateState` of the associated memory object
- `recoverObject` is called when an inconsistent transactional object is accessed after a crash failure



McGill

# Recovery Rules [BCF97]

- Undo Recovery Rule (or Write Ahead Logging):
  - All information necessary for undoing the changes made to the state of an object during execution of one of its operations must be written to the log before the object's state is propagated to its associated storage unit.
- Redo Recovery Rule (or Commit Rule):
  - All information necessary for redoing the changes made to the states of all objects modified during a transaction must be written to the log before the transaction commits.



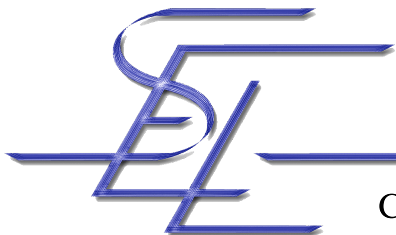
McGill

# Undo/NoRedo Example Code (1)

- PreOperation
  - Pin the transactional object
  - If physical logging is used
    - If this is the first operation modifying the state of this object invoked during this transaction
      - Save before-image in memory
  - If logical logging is used
    - Save undo information for this operation in memory
- PostOperation
  - Unpin the transactional object

# Undo/NoRedo Example Code (2)

- PropagateObject
  - Write all undo information for this object from memory to stable storage
- AbortTransaction
  - For all modified objects
    - Undo the changes to the object in memory by applying the gathered undo information in memory
    - Optional: If object was previously propagated, propagate the (old) state of the object to the associated storage unit and discard undo information for this transaction
    - Send abort notification to the concurrency control of the object
  - Log transaction abort



# Undo/NoRedo Example Code (3)

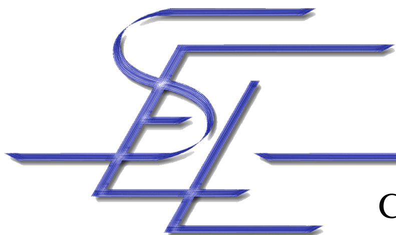
- CommitTransaction
  - If we are in a top-level transaction
    - For all accessed objects
      - If the object is dirty (i.e. has been modified)
        - Propagate the objects state to the associated storage unit
    - Log transaction commit (*point of no return*)
    - For all accessed objects
      - Send commit notification to the concurrency control of the object
      - Delete undo information for this transaction from memory and, if necessary, from stable storage
  - Else
    - Pass undo information of the object to parent transaction

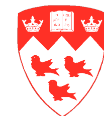
# Transaction Support Interfaces

- Allow a programmer to clearly mark transaction boundaries
- Monitor exceptions crossing the transaction boundary
- Pass control to the framework whenever an operation on a transactional object is invoked
- The elegance of the interface is highly programming language dependent

# Transaction Identification

- If the programming language does not allow associating data with threads, then transaction identifiers must be made explicit
  - Starting a transaction hands back a TID
  - The TID must be passed as a parameter to all calls that involve the transaction support
    - Aborting / committing the transaction
    - Invoking operations on transactional objects
  - Error-prone
    - Allows working in two transactions simultaneously





McGill

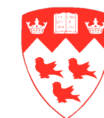
# Procedural Interface

- ```
  void beginTransaction();
  or: tid beginTransaction();
  ```
- ```
  void abortTransaction();
  or: abortTransaction(tid t);
  ```
- ```
  void commitTransaction();
  or: commitTransaction(tid t);
  ```

Flexible

Error-prone

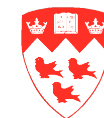
Relies on programmer to catch exceptions!



McGill
  ```

# Associating Threads with Transactions

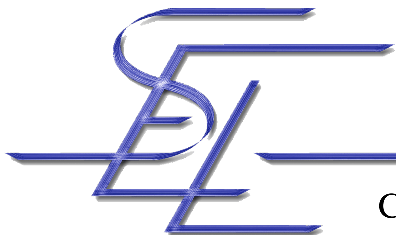
```
public class TransactionContext {
  static ThreadLocal myContext =
    new InheritableThreadLocal();
  static public void setTransaction(Transaction t) {
    myContext.set(t);
  }
  static public Transaction getTransaction() {
    return (Transaction) myContext.get();
  }
  private TransactionContext() {
  }
}
```



McGill

# Using the Procedural Interface

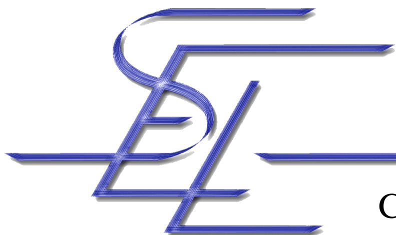
```
void transfer (Account source,
   Account dest, int amount) {
   ProceduralInterface.beginTransaction();
   try {
      source.withdraw(amount);
      dest.deposit(amount);
      ProceduralInterface.commitTransaction();
   } catch (notEnoughFundsException e) {
      ProceduralInterface.abortTransaction();
   }
}
```

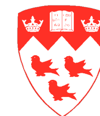


# Object-Based Interface

```
package Object_Based_Transaction_Interface is
   type Transaction is limited private;
   procedure Commit_Transaction (T : in out Transaction);
private
   type Transaction is new
      Ada.Finalization.Limited_Controlled with record
      Committed : Boolean := False;
   end record;
   procedure Initialize (T : in out Transaction);
   procedure Finalize (T : in out Transaction);
end Object_Based_Transaction_Interface;
```

Constructor and destructor begin rsp. commit/abort the transaction





McGill

# Using the Object-Based Interface

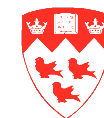
```
declare
  T : Transaction;
begin
  -- perform work on behalf of the transaction
  -- by calling operations on objects
  Commit_Transaction (T);
exception
  when ... ⇒
    -- handle internal exceptions
    Commit_Transaction (T);
  when ... ⇒
    -- raise an external exception
end;
```

# Aspect-Oriented Interface (1)

```
public abstract aspect TransactionalMethod {

  abstract public pointcut Method();

  void around() : Method () {
       ProceduralInterface.beginTransaction();
       boolean aborted = false;
       try {
          proceed();
       } catch (TransactionException e) {
           ProceduralInterface.abortTransaction();
           aborted = true; throw e;
       } finally {
          if (!aborted) {
             ProceduralInterface.commitTransaction();
          }
       }
    }
}
```



McGill

# Aspect-Oriented Interface (2)

```
class AccountManager {
   public void transfer (Account source, Account
     dest, int amount) {…}
}

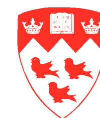
aspect MakeAccountMgrMethodsTransactional
   extends TransactionalMethod {

   public pointcut Method () :
      call (public * AccountManager.*(..));
}
```

*AspectJ* code making all methods of instances of the `AccountManager` class execute within a transaction

# Transactional Objects

- Operations called by the application programmer
  - Application-defined operations
  - Creation, restoration and deletion operations
- Operations called by the transaction framework
  - Loading and saving the state to the associated storage
  - Provide semantic information for concurrency control and recovery (read/write, commutativity table)
  - If logical logging is used, inverse operations must be designated for every operation, as well as means to load and save operation invocations



McGill

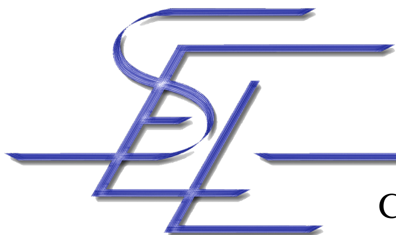
# Transactional Wrapper

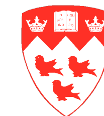
| ApplicationObject |
|---|
| |
| methods(..) |

| TransactionalObject |
|---|
| |
| methods(..)<br>create(NonVolatileParameter)<br>restore(NonVolatileParameter)<br>delete()<br>abortTransaction()<br>commitTransaction() |

TransactionalObject inherits from ApplicationObject.

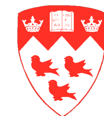
Perform the necessary calls to the framework, then call original method





# Interaction with OPTIMA

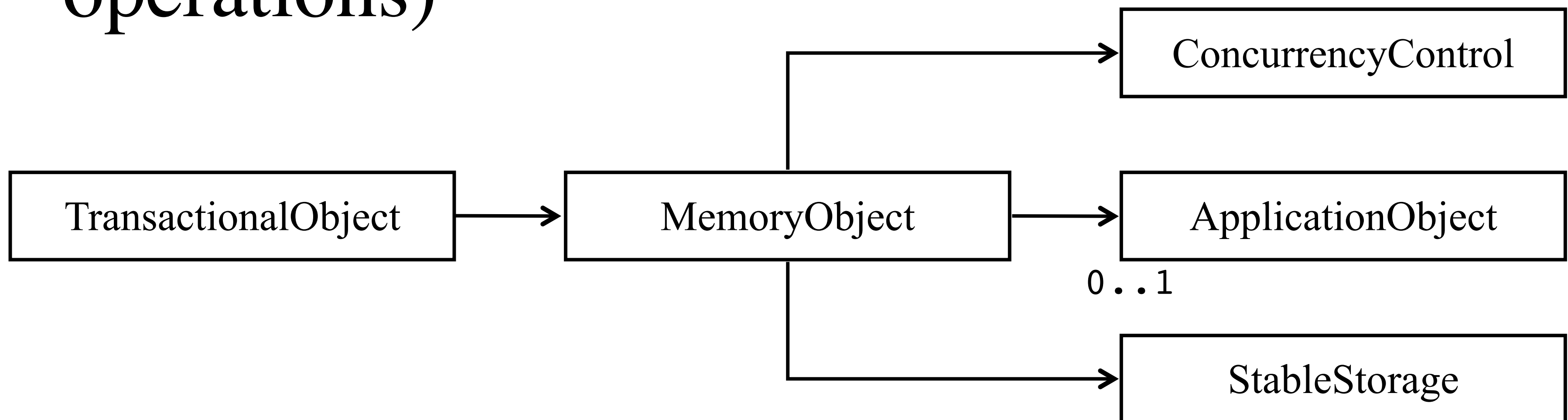
1. Concurrency Control Prologue
   Call preOperation of the concurrency control
2. Recovery Prologue
   Call preOperation of the recovery manager
3. Method Execution
4. Recovery Epilogue
   Call postOperation of the recovery manager
5. Concurrency Control Epilogue
   Call postOperation of the concurrency control



McGill

# Ada Transactional Objects

- The transactional object wrapper has to be written by the application programmer
- Could be automated using a preprocessor (except for semantic information on operations)



McGill

# Aspect-Oriented Interface (1)

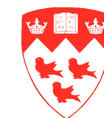
```
aspect TransactionalObject
   pertarget(Method) {

   pointcut Method() :
      call(public * Account.*(..));

   private final RecoveryManager
      myRecoveryManager = …;

   private final ConcurrencyControl
      myConcurrencyControl = new …;

   // more stuff to be added here
}
```



McGill

# Aspect-Oriented Interface (2)

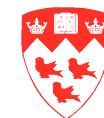
```
aspect TransactionalObject
  pertarget(Method) {

  pointcut Method(Object o) : call
    (public * Account.*(..)) && target(o);

  // code from previous slide

  around() : Method(Object o) {
    Transaction t =
      TransactionContext.getTransaction();
    if (t != null) {
      myConcurrencyControl.preOperation(o,t);
      myRecoveryManager.preOperation(o,t);
    }
    proceed();

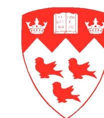
    if (t != null) {
      myRecoveryManager.postOperation(o,t);
      myConcurrencyControl.postOperation(o,t);
    }
}
```



McGill

# Customization

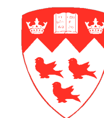
- Transactions are used in different application domains with different requirements
- Customize
  - Caching
  - Recovery Strategy
  - Storage used for Persistence
  - Concurrency Control
- Without customization, worst-case assumptions must be made, which might lead to poor performance



McGill

# OPTIMA Customization

- At startup
  - Cache manager
  - Recovery manager
  - Stable storage used for storing the log
- Per object
  - Storage used to store the object's state
  - Concurrency control (?)
- Per method
  - Semantic information



McGill

# Specifying Semantic Information

- Programmer writes

```
public class GetBalanceInfo extends OperationInfo {

   public boolean backwardCommutesWith (OperationInfo op)
     { return (op instanceof GetBalanceInfo);
   }
}
```

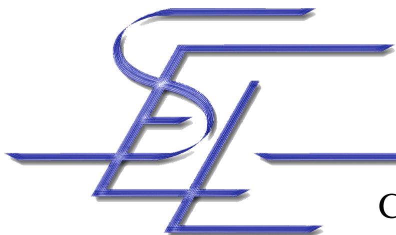
- The framework defines

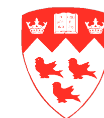
```
public interface CustomizedMethods {

   public OperationInfo getOperation
      (String name, JoinPoint jp)
      throws MethodCustomizationException;
}
```

# Applying Customization

```
public aspect AccountMethodAspect {
    declare parents:
        Account implements CustomizedMethods;
    public OperationInfo
        Account.getOperation
        (String name, Joinpoint jp) throws
        MethodCustomizationException {
        if (name.equals(“getBalance”)) {
            return new GetBalanceInfo();
        } else if { …
        }
    }
}
```



McGill

# References (1)


- [K+01]
  Kienzle, J.; Jiménez-Peris, R.; Romanovsky, A.; Patiño-Martinez, M.: “Transaction Support for Ada”, in Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001, pp. 290 – 304, Lecture Notes in Computer Science 2043, Springer Verlag, 2001.
- [GHJV95]
  Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison Wesley, Reading, MA, USA, 1995.
- [LS79]
  Lampson, B. W.; Sturgis, H. E.: “Crash Recovery in a Distributed Data Storage System”. Technical report, XEROX Research, Palo Alto, June 1979.
- [KR02]
  Kienzle, J.; Romanovsky, A.: “A Framework Based on Design Patterns for Providing Persistence in Object-Oriented Programming Languages”, IEE Software Engineering Journal 149(3), pp. 77 - 85, June 2002.

# References (2)


- [CKS01]
  Caron, X.; Kienzle, J.; Strohmeier, A.: “Object-Oriented Stable Storage based on Mirroring”. In Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001, pp. 278 – 289, Lecture Notes in Computer Science 2043, Springer Verlag, 2001.
- [BCF97]
  Besancenot, J.; Cart, M.; Ferrié, J.; Guerraoui, R.; Pucheral, P.; Traverson, B.: Les Systèmes Transactionnels: Concepts, Normes et Produits. Editions Hermes, Paris, France, 1997.