# RATIONAL PURIFY

przygotował: Krzysztof Jurczuk
Politechnika Białostocka
Wydział Informatyki Katedra Oprogramowania
ul. Wiejska 45A
15-351 Białystok

**STRESZCZENIE:** Dokument przestawia podstawową funkcjonalność narzędzia IBM Rational Purify. Omawia problem analizy czasu wykonania oraz przedstawia możliwości prezentowanego narzędzia. Prezentuje także użycie Rational Purify przy pracy z kodem napisany w C++ i JAVA oraz z kodem zarządzanym platformy .NET. Porusza także wewnętrzną zasadę działania przedstawianego oprogramowania.

## 1. WPROWDZENIE

Rational Purify jest narzędziem wspomagającym proces tworzenia oprogramowania. Pozwala na wczesne i łatwe wykrywanie błędów kompilacji oraz "wycieków pamięci" (ang. *memory leaks*). Pozwala to na odpowiednio wczesne wykrycie nieporządnych błędów w oprogramowaniu, co pozwala na szybszą oraz bardziej niezawodną pracę zespołów programistycznych. Błędy kompilacji i "wycieki pamięci" są jednymi z najbardziej trudnych błędów do wykrycia. Skutki jakie mogą powodować są trudne do przewidzenia. Poza tym nie zawsze musimy na nie trafić podczas każdego uruchomienia aplikacji. Bardzo często pozostają nieodkryte przez długi czas działa oprogramowania. Dlatego też bardzo ważnym elementem podczas tworzenia aplikacji jest korzystanie z narzędzi wspomagających proces detekcji błędów, szczególnie błędów wykonania (ang., *runtime analysis*).

Rational Purify umożliwia między innymi:

- szybkie i wszechstronne wyszukiwanie błędów czasu wykonania
- detekcje błędów bez posiadania kodu źródłowego
- wraz z użyciem narzędzia IBM Rational PureCoverage określenie miejsc w kodzie, które nie zostały przetestowane

## 2. BŁĘDY WYKRYWANE PRZEZ RATIONAL PURIFY

Rational Purify pozwala wykryć między innymi następujące błędy:

| | | |
|---|---|---|
| Array Bounds Read | ABR | czytanie z pamięci spoza zakresu tablicy |
| Array Bounds Write | ABW | pisanie do pamięci spoza zakresu tablicy |
| Beyond Stack Read | BSR | czytanie powyżej bieżącego wskaźnika stosu |
| Beyond Stack Write | BSW | pisanie powyżej bieżącego wskaźnika stosu |
| Freeing Freed Memory | FFM | próba dealokacji już wcześniej zwolnionej pamięci |
| Freeing Invalid Memory | FIM | próba dealokacji pamięci, której użytkownik nie może zwalniać, np pamięć zarezerwowana na kodu programu |
| Free Memory Read | FMR | czytanie z pamięci, która została zwolniona |
| Free Memory Write | FMW | pisanie do pamięci, która została zwolniona |

| | | |
|---|---|---|
| Invalid Pointer Read | IPR | czytanie z nieprawidłowej pamięci, np. z pamięć nieistniejącej w pamięci adresowej, pamięci z pierwszych 64K pamięci adresowej |
| Invalid Pointer Write | IPW | pisanie do nieprawidłowej pamięci, np. do pamięć nieistniejącej w pamięci adresowej, pamięci z pierwszych 64K pamięci adresowej |
| Memory Allocation Failure | MAF | Niepowodzenie alokacji pamięci |
| Memory Leak | MLK | "wycieki pamięci" |
| Null Pointer Read | NPR | czytanie z pamięci o adresie 0 |
| Null Pointer Write | NPW | pisanie do pamięci o adresie 0 |
| Uninitialized Memory Read | UMR | czytanie z niezainicjowanej pamięci |
| Uninitialized Memory Copy | UMC | kopiowanie niezainicjowanej pamięci |
| Handle In Use | HIU | niezwolnienie zasobów związanych z uchwytem |

W poniższym kodzie zostały pokazane przykładowe przyczyny wyżej wymienionych błędów:
(UWAGA!!! Prostota błędów w poniższym kodzie wynika z ich dydaktycznego przeznaczenia.)

```
//made by Krzysztof Jurczuk
//rationalBugsExample.cpp : Defines the entry point for the console application.

#include "stdafx.h"
#include <iostream>
using namespace std;

class CTest{
private:
	int* pointer;
public:
	int* beyonStack(){
		int i;
		return &i;
	}

	int getPointer(){
		return *pointer;
	}

};

int main(int argc, char* argv[]){

	////////////////////////////////////////////////////////////
	CTest test;
	int* i = test.beyonStack();
	cout << *i;                                    //BSR
	*i = 4;                                        //BSW

	////////////////////////////////////////////////////////////
	const int MAX = 10;
	int* table1 = new int[ MAX ];
	int* table2 = new int[ MAX ];

	for( int k = 0; k <= MAX; k++ )
		table1[ k ] = table2[ k ];             //ABW, ABR
```

```
        ////////////////////////////////////////////////////////////
        int* j = new int();

        delete j;
        delete j;                                           //FFM

        cout << *j;                                         //FMR
        *j = 4;                                             //FMW

        j = NULL;
        cout << *j;                                         //NPR, EXU
        *j = 4;

        return 0;
}
```

UWAGA!!! Wykonywanie powyższego kodu może powodować nieprzewidziane skutki.

Wynik uzyskany po wykonaniu wypowyższego kodu przy użyciu Rational Purify:

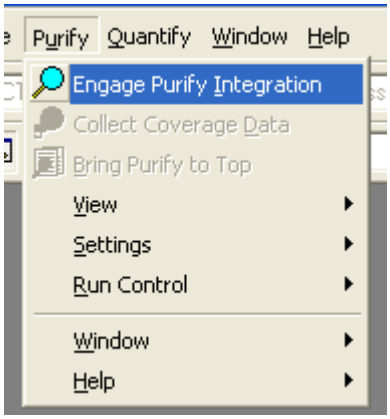
```
Starting Purify'd E:\test\Debug\test.exe at 12/17/2006 11:54:47
Starting main
BSR: Beyond stack read in main {1 occurrence}
    Reading 4 bytes from 0x0013f1a0 (top of stack is at 0x0013f1b0))
    Address 0x0013f1a0 points into a thread's stack
    Address 0x0013f1a0 is 204 bytes past the start of local variable 'j' in main
    Thread ID: 0xeb0
    Error location
        main            [E:\test\test.cpp:29]
            ////////////////////////////////////////////////////////////
            CTest test;
            int* i = test.beyonStack();
            cout << *i;                 //BSR
            *i = 4;                     //BSW


        mainCRTStartup [crt0.c:206]
BSW: Beyond stack write in main {1 occurrence}
ABR: Array bounds read in main {1 occurrence}
ABW: Array bounds write in main {1 occurrence}
FFM: Freeing freed memory in delete(void *) {1 occurrence}
FMR: Free memory read in main {1 occurrence}
FMW: Free memory write in main {1 occurrence}
NPR: NULL pointer read in main {1 occurrence}
EXU: Unhandled exception in main {1 occurrence}
    Exception code: 0xc0000005 [Error: access violation reading from 0x00000000]
    Exception address: main             [E:\test\test.cpp:51]
    Filter: mainCRTStartup [crt0.c:212]
    Exception location
        main            [E:\test\test.cpp:51]
            *j = 4;                     //FMW

            j = NULL;
            cout << *j;                 //NPR, EXU
            *j = 4;

            return 0;
        mainCRTStartup [crt0.c:206]
Summary of all memory leaks... {80 bytes, 2 blocks}
Exiting with code -1073741819 (0xc0000005)
Program terminated at 12/17/2006 11:54:49
```

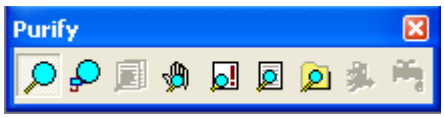
## 3. PODSTAWY OBSŁUGI

Rational Putify może zostać zintegrowany z wieloma środowiskami. Wersja 6.0 automatycznie integruje się z MS Visual 6.0, natomiast wersja 7.0 z MS Visual .NET. Każda z nich posiada możliwość zintegrowania z Eclipsem poprzez dołączone pluginy. Co więcej aplikacje w wersji skompilowanej mogą zostać sprawdzone poprzez ich uruchomienie w Rational Purify, który nie jest zintegrowany z żadnym środowiskiem (tzw. z ang. *standalone application*). Taki proces uruchomienia jest identyczny jak przy omówionej poniżej obsłudze JAVY (patrz punkt 3b)

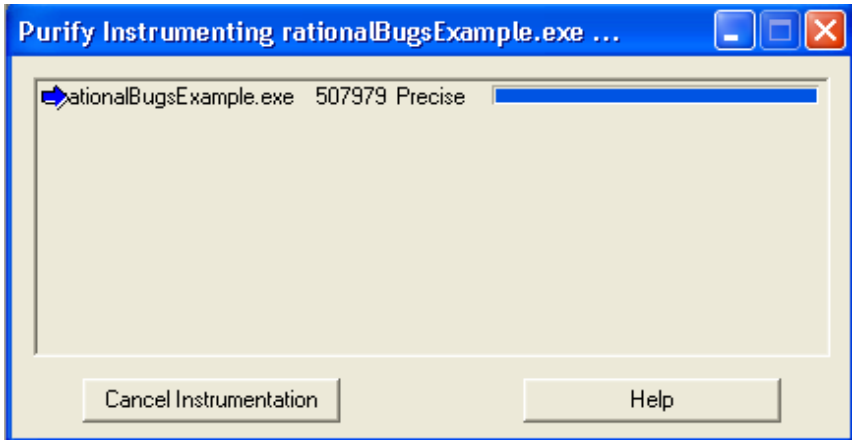
### a) MS Visual C/C++

Przed uruchomieniem aplikacji należy powiadomić zintegrowany Rational Purify o konieczności zbierania informacji o poprawności kodu poprzez wybranie opcji Engaged Purify, z menu Purify lub odpowiedniego toolbaru:

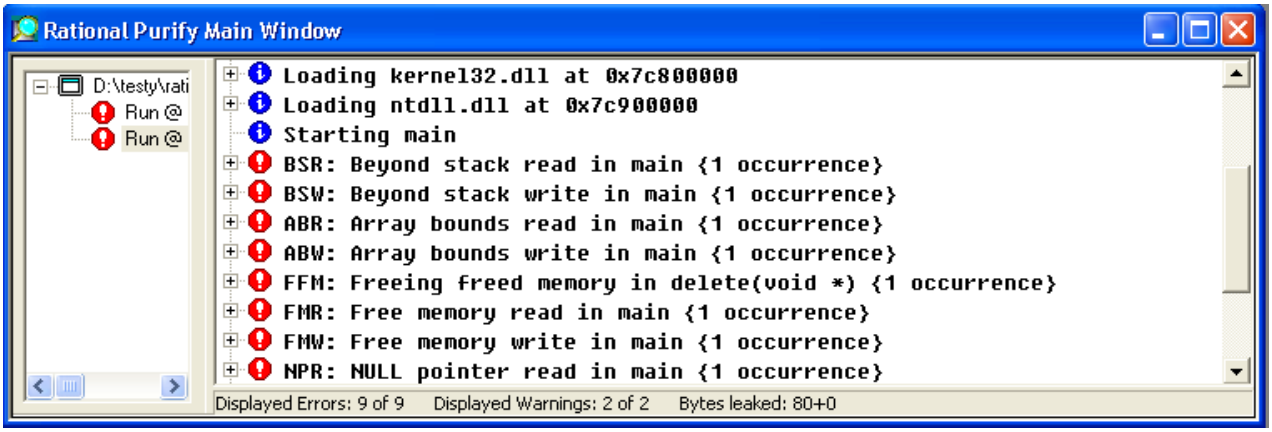


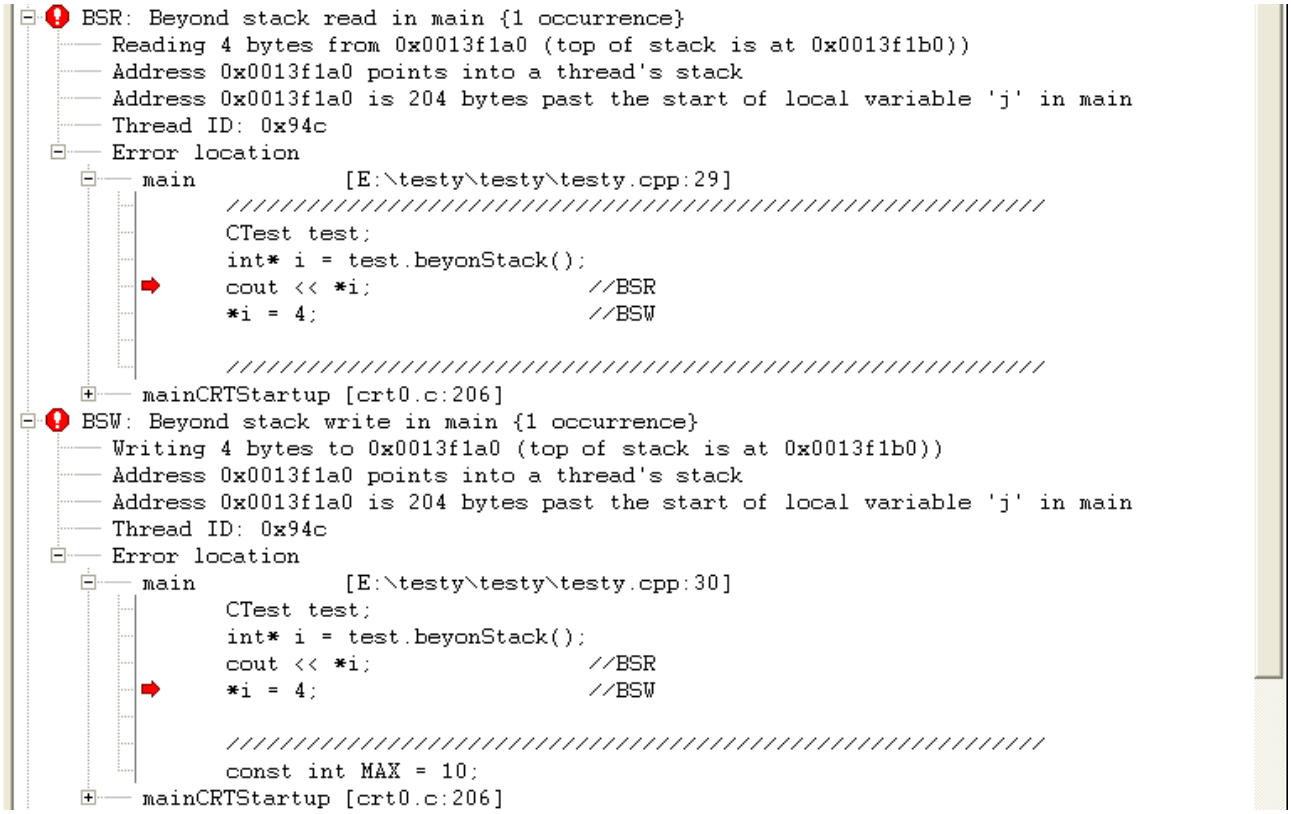

Zapewnia do dołączenie podczas kompilacji odpowiednich danych pozwalających na zbieranie potrzebnych informacji podczas wykonania:

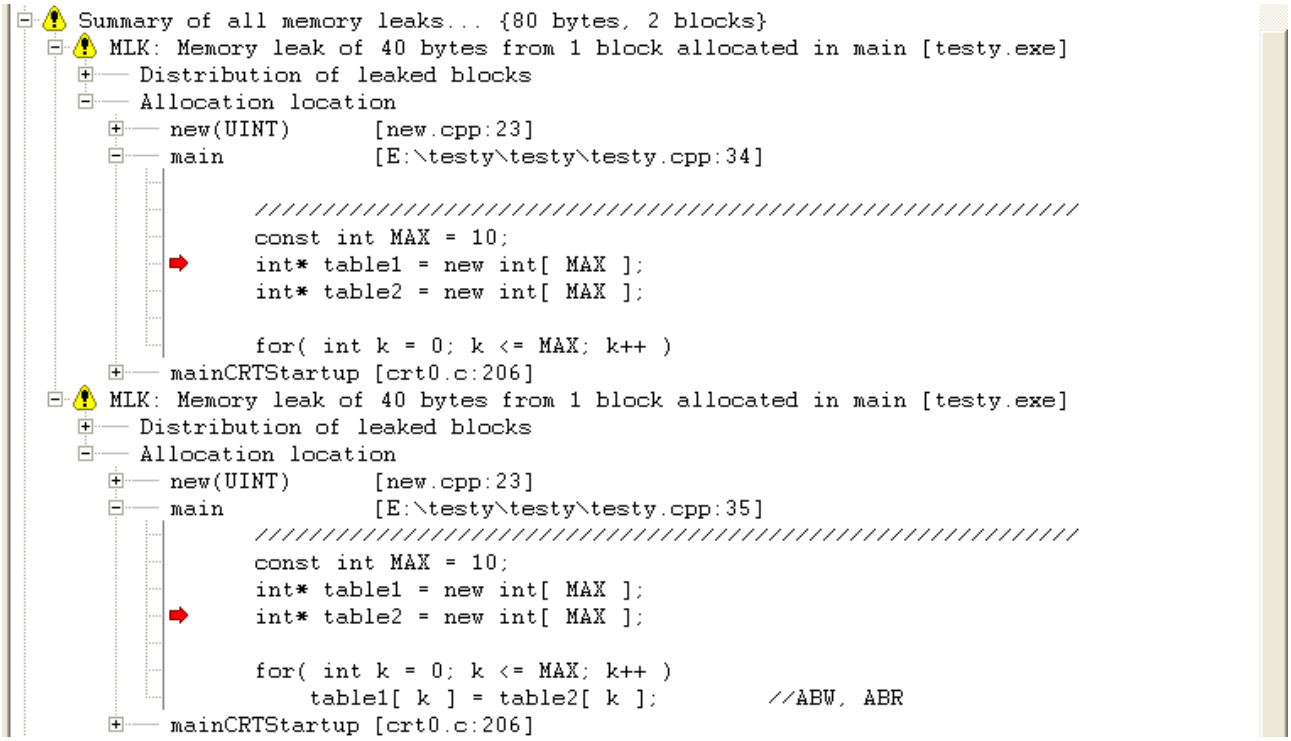



Po wykonaniu aplikacji zostaje przedstawiony raport z ewentualnymi błędami:



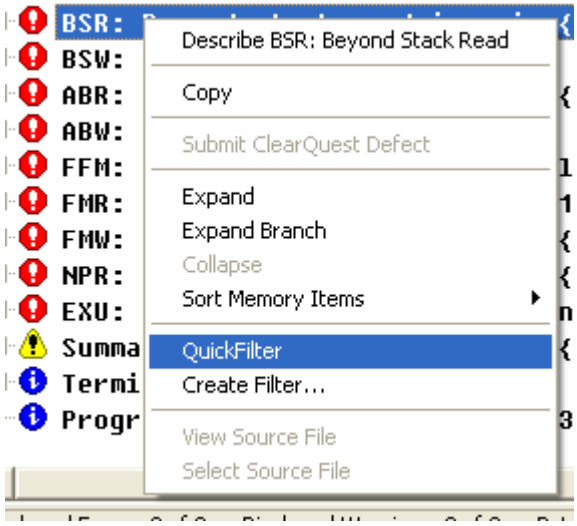
Wszystkie informacje w raportach są umieszczane w postaci drzewa, w którym wraz z większym zagłębieniem znajdziemy coraz więcej szczegółowych informacji na temat wybranego błędu:

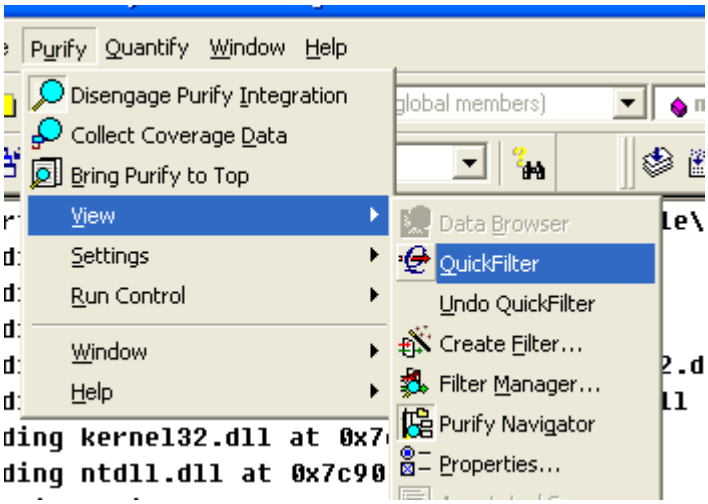



Umożliwia to także oglądanie kodu, który jest powodem danego błędu oraz bezpośrednie przeniesienie się do kodu źródłowego poprzez dwukrotne kliknięcie.

Jeśli chcemy wyfiltrować pewne rodzaje błędów możemy użyć tzw. "filtrów szybkich" poprzez wybór, z menu podręcznego lub z menu głównego, opcji "Quick Filter":




Umożliwia to pozostawienie najważniejszych błędów przy braku czasu na analizę wszystkich. Możemy także stworzyć własne filtry poprzez opcje "Create Filter". Istnieje także możliwość porównania raportów z kolejnych wykonań programu przy użyciu opcji: "Run Controls ->Compare Runs".

Uzyskane wyniki mogą zostać zapisane dla późniejszej analizy, czy też tworzenia raportów błędów tworzonego oprogramowania. Można także poddawać analizie aplikacje skompilowane (patrz. punkt 3b). Prezentowana aplikacja może być także używana podczas debugowania, wystarczy wybrać opcję "Settings->Break on Error". Gdy testujemy oprogramowanie wielomodułowe (np. wiele plików *.dll), można wyeliminować wybrane moduły ("Settings->Executable Settings" zakładka "Power Check" i opcja All modules – configures).

**b) JAVA**

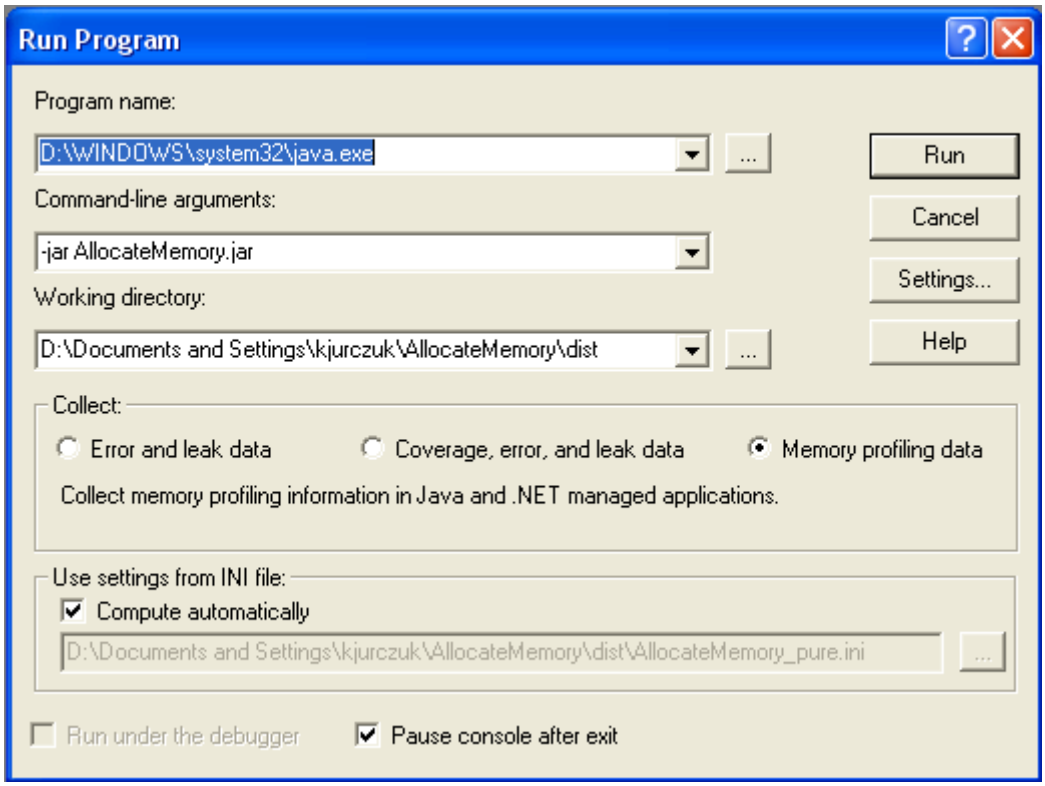
Sytuacja przy programowaniu w języku JAVA wydaje się być o wiele prostsza ze względu na to, iż w większości przypadków nie musimy martwić się o błędy związane z pamięcią, ponieważ jej zwalnianiem zajmuje się Garbage Collector. Jednak bardzo często nie tylko początkujący programiści potrafią nieświadomie zapominać o powiadomieniu, że pewnie zasoby pamięciowe mogą być zwolnione. Niepotrzebna pamięć zabiera nam zasoby systemowe ograniczając tym samym pulę dostępnej, wolnej pamięci. Dlatego też Rational Purify pozwalana prześledzić jakie porcje pamięci są alokowane przed konkretne elementy w aplikacji. Ułatwia to znalezienie miejsc, które rezerwują zbyt dużo pamięci oraz miejsc, które zbyt długo przetrzymują niewykorzystywaną, ale nadal zarezerwowaną pamięć. Pozwala to zoptymalizować działanie tworzonego produktu, czyniąc go tym samym mniej "pamięcio-żernym".

Podstawowymi błędami popełnianymi przez programistów JAVA, które są powodem niezwalniania już niepotrzebnej pamięci są:

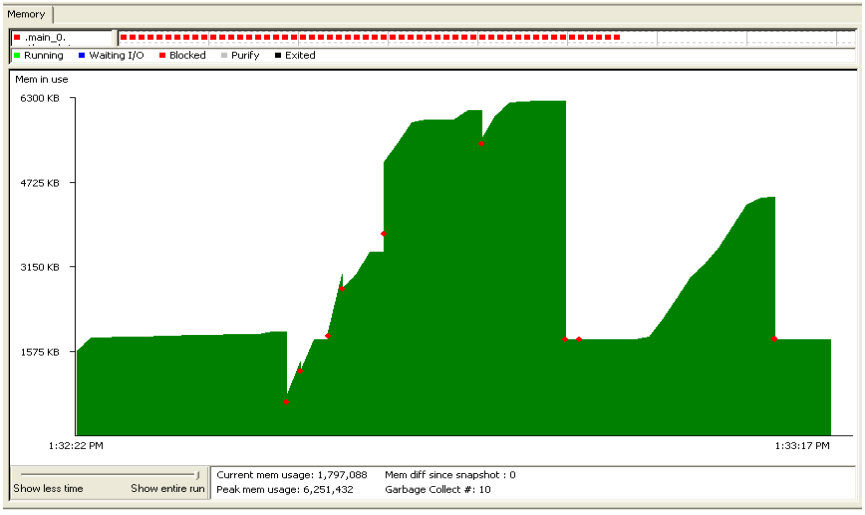
- dodawanie obiektów do tablicy i pozostawienie ich tam, gdy są już nam niepotrzebne przy dalszym używaniu innych elementów tablicy (analogicznie jeśli chodzi o struktury dynamiczne jak Vector, czy też List)
- przetrzymywanie referencji w długo trwających wątkach, podstawienie pod nie NULL nie pomoże, zostaną dopiero zwolnione w momencie zakończenia wątku

Aplikacje napisane w języku JAVA można testować na wiele sposobów, chociażby poprzez zintegrowanie posiadanego środowiska z Rational Purify lub też niezależnie od platformy programistycznej używając samego IBM Rational Purify (tzw. z ang. *standalone application*). Taki też sposób został przestawiony poniżej.

W oknie powitalnym wybieramy opcję "Run". Następnie wybieramy do analizy plik *.jar z naszą aplikacją lub też plik *.class, który zawiera klasę z funkcją uruchomieniową (mimo iż na poniższym rysunku nazwa programu znajduje się w drugim polu należy ją podać w pierwszym, dalej już Rational Purify ustawi sam parametry wywołania tak jak jest to widoczne poniżej):

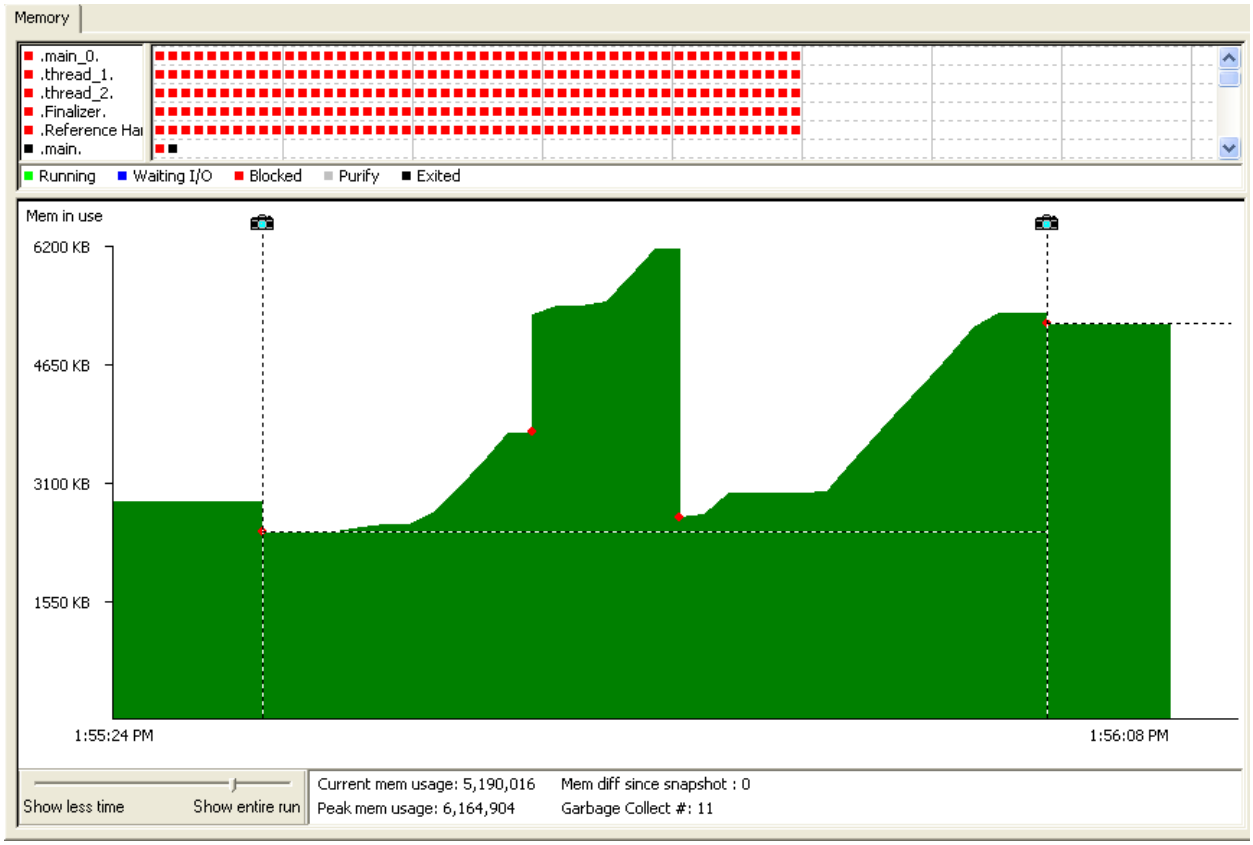


Po uruchomieniu aplikacji pojawi się nam analizator wykonania, na którym jest przedstawiony stan pamięci. Czerwone kropki oznaczają miejsca, w których działał Gargabe Collector. Na osi poziomej jest umieszczony czas, natomiast na osi pionowej stan pamięci.

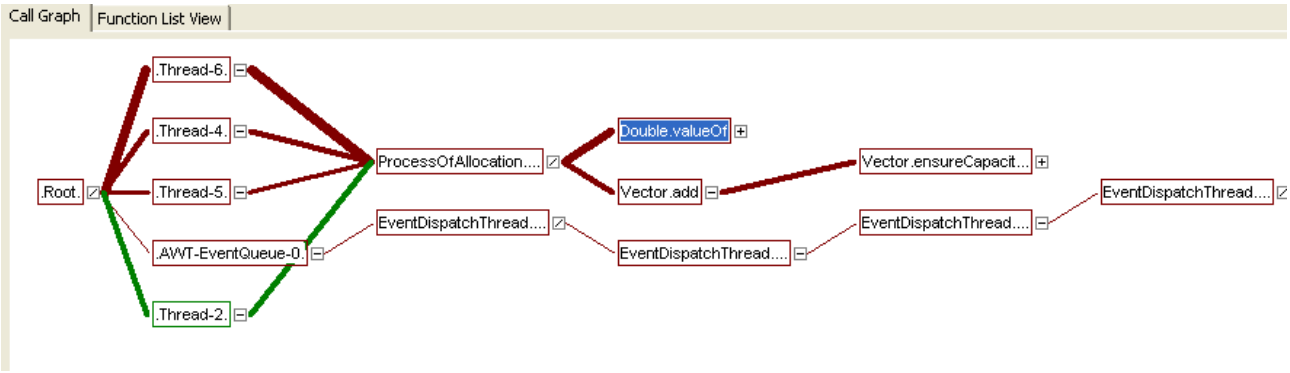



Standardowy proces analizy wygląda następująco:

- po uruchomieniu aplikacji czekamy, aż użycie pamięci ustabilizuje się
- wykonujemy zrzut stanu pamięci przy użyciu opcji "Snapshot memory profiling data" z toolbaru 
- następnie wykonujemy aplikację z użyciem opcji, którą podejrzewany o nieprawidłowe zarządzanie wykorzystywaną pamięcią, następnie wracamy do stanu wyjściowego i znów wykonujemy zrzut stanu pamięci
- jeśli różnica zajętości pamięci jest znaczna (mimo wywołania działania Garbage Collector'a, poprzez opcję "Gargabe collect" z toolbaru  ) może to świadczyć o nieprawidłowej obsłudze pamięci i stanowi to podstawę do analizy kodu odpowiedzialnego za sprawdzaną funkcjonalność



Można także porównać zrobione zrzuty stanu pamięci. Wystarczy zaznaczyć jeden z nich i użyć opcji "Compare runs" z toolbaru Δ. Jeśli istnieje możliwość wybieramy jeden z poprzednich zrzutów. Jako wynik zobaczymy graf porównania wykonań:




Zielone ścieżki oznaczają polepszenie stanu pamięci, natomiast czerwone większą zajętość pamięci. Więcej informacji można także uzyskać poprzez umieszczenie myszki nad elementami w grafie wywołań. W zakładce "Function List View" znajdziemy porównania wykonań konkretnych funkcji:

Call Graph | Function List View

| Method | Calls (New) | Calls (Base) | Current method bytes allocated (Diff) | Class | Source File |
|---|---|---|---|---|---|
| .AWT-EventQueue-0. | 0 | 0 | 0 | (None) | (None) |
| .AWT-Shutdown. | 0 | 0 | 0 | (None) | (None) |
| .AWT-Windows. | 0 | 0 | 0 | (None) | (None) |
| .DestroyJavaVM. | 0 | 0 | 0 | (None) | (None) |
| .Finalizer. | 0 | 0 | 0 | (None) | (None) |
| .Java2D Disposer. | 0 | 0 | 0 | (None) | (None) |
| .main. | 0 | 0 | 0 | (None) | (None) |
| .main_0. | 0 | 0 | 0 | (None) | (None) |
| .Reference Handler. | 0 | 0 | 0 | (None) | (None) |
| .Root. | 0 | 0 | 0 | (None) | (None) |
| .Secondary finalizer. | 0 | 0 | 0 | (None) | (None) |
| .Signal Dispatcher. | 0 | 0 | 0 | (None) | (None) |
| .thread_1. | 0 | 0 | 0 | (None) | (None) |
| .thread_2. | 0 | 0 | 0 | (None) | (None) |
| .thread_3. | 0 | 0 | 0 | (None) | (None) |
| .thread_4. | 0 | 0 | -136 | (None) | (None) |
| .Thread-2. | 0 | 0 | 0 | (None) | (None) |
| .Thread-3. | 0 | 0 | 0 | (None) | (None) |

Istnieje też możliwość uzyskania większej ilości informacji na temat danej funkcji poprzez dwukrotne kliknięcie na jej nazwie.

Podobnie jak przy testowaniu aplikacji w niezarządzanym C/C++ możemy tworzyć filtry poprzez opcję "Filter Manager" z menu podręcznego. Uzyskane dane można zapisać np. w celu tworzenia raportów.

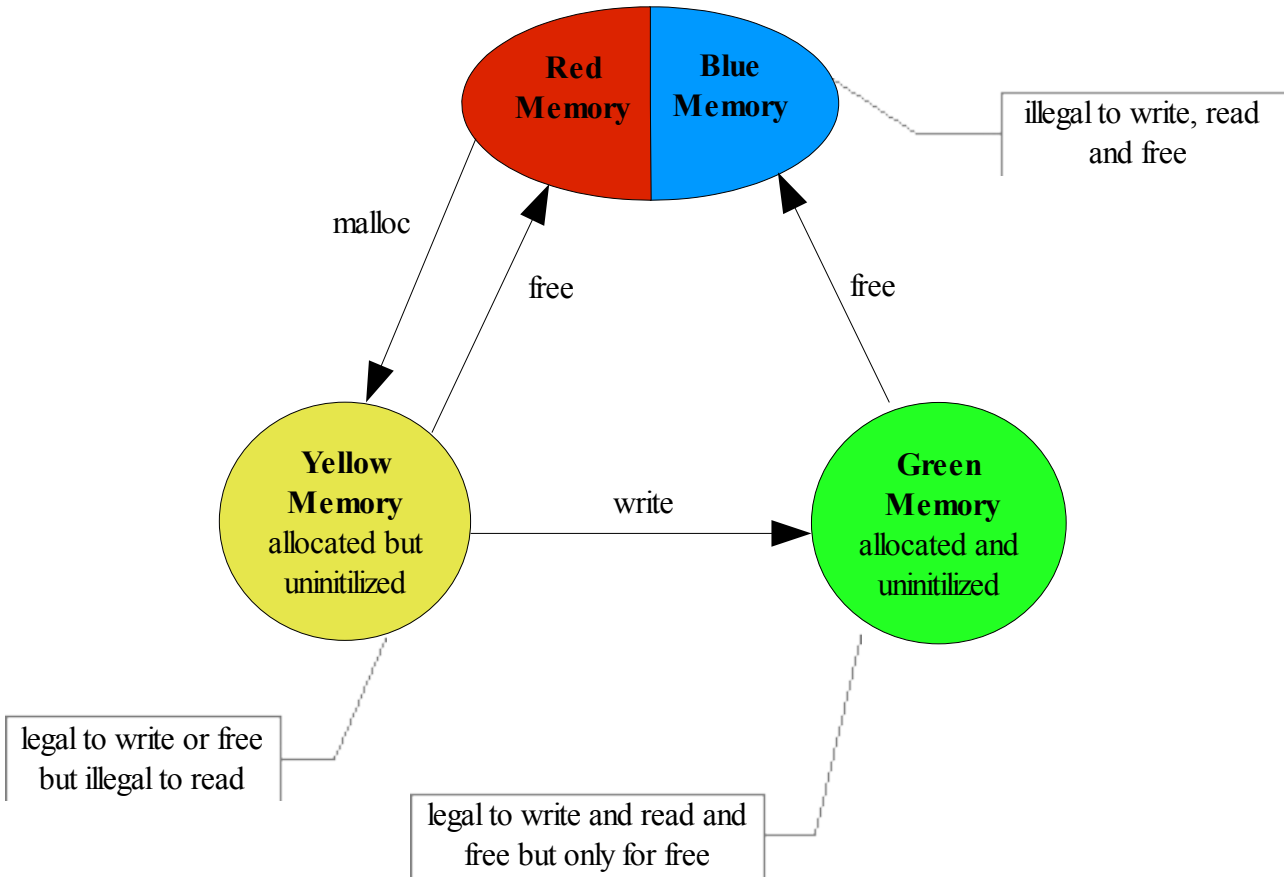
**c) MS Visual .NET (managed code)**
Faza testowania aplikacji napisanej w kodzie zarządzanym platformy .NET wyglądanie identycznie jak przy kodzie w języku JAVA (patrz punkt 3b). Natomiast proces uruchomienia identycznie jak przy kodzie niezarządzanym C/C++ (patrz punkt 3a).

## 4. WEWNĘTRZNA ZASADA DZIAŁANIA RATIONAL PURIFY

Purify monitoruje wszystkie operacje związane z pamięcią w testowanej aplikacji i sprawdza czy są poprawne. Zapamiętuje informacje na temat pamięci:

- niezaalkowanej
- zaalokowanej niezainicjalizowanej
- zaalkowanej zainicjalizowanej
- zwolnionej.

Służy do tego tablica, w której śledzony jest każdy bajt pamięci. Na każdy bajt w tablicy są przeznaczone 2 bity. Pierwszy mówi, czy pamięć jest zaalkowana, natomiast drugi czy jest zainicjalizowana. Pozwala to na zapamiętanie maksymalnie 4 stanów: czerwonego, żółtego, zielonego oraz niebieskiego:




IBM Rational Purify przy każdej operacji związanej z pamięcią, sprawdza czy dana operacja jest dozwolona. Jeżeli zauważa nieprawidłową operacje dodaje błąd do raportu.

Pamięć żółta - pamięć otrzymywana po wykonaniu „malloc” lub „new”; jest to pamięć zaalokowana, ale niezainicjowana, zatem nie możemy na niej wykonywać operacji czytania
Pamięć zielona - pamięć zaalkowana oraz zainicjalizowana

Pamięć czerwona – pamięć zwolniona, z której nie możemy korzystać do momentu następnej alokacji
Pamięć niebieska – pamięć zwolniona, która wcześniej była zaalkowana oraz zainicjalizowana, ale już w tym momencie nie można z niej korzystać, do momentu kolejnej alokacji.

IBM Rational Purify oprócz błędów związanych z pamięcią dynamiczną, potrafi wykrywać niepoprawne instrukcje na zmiennych globalnych oraz statycznych. Wykonuje to poprzez umieszczenie „stref czerwonych” (czyli pamięci o kolorze czerwonym) wokół pamięci przydzielonej dla danych zmiennych. Nie zapewnia to zawsze wykrycia błędów, ale na pewno pozwala na wykrycie korzystania z pamięci niezainicjalizowanej lub niezaalkowanej.