Design-Time Performance Testing

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Ian Hopkins

Permission to Use

Abstract

Software designers make decisions between alternate approaches early in the development of a software application and these decisions can be difficult to change later. Designers make these decisions based on estimates of how alternatives affect software qualities. One software quality that can be difficult to predict is performance, that is, the efficient use of resources in the system. It is particularly challenging to estimate the performance of large, interconnected software systems composed of components. With the proliferation of class libraries, middle-ware systems, web services, and third party components, many software projects rely on third party services to meet their requirements. Often choosing between services involves considering both the functionality and performance of the services. To help software developers compare their designs and third-party services, I propose using performance prototypes of alternatives and test suites to estimate performance trade-offs early in the development cycle, a process called *Design-Time Performance Testing (DTPT)*.

Providing software designers with performance evidence based on prototypes will allow designers to make informed decisions regarding performance trade-offs. To show how DTPT can help inform real design decisions. In particular: a process for DTPT, a framework implementation written in Java, and experiments to verify and validate the process and implementation. The implemented framework assists when designing, running, and documenting performance test suites, allowing designers to make accurate comparisons between alternate approaches. Performance metrics are captured by instrumenting and running prototypes.

This thesis describes the process and framework for gathering software performance estimates at design-time using prototypes and test suites.

Acknowledgements

I would like to express my gratitude to the following people who have help me in writing this thesis:

- My Supervisor: Dr. Kevin Schneider, for his guidance, encouragement, and enthusiasm for this project.
- My Committee: Dr. Chris Dutchyn, Dr. Nate Osgood, and Dr. Igor Morozov.
- My Family: For their encouragement and support of my education and research efforts.
- My Lab Cohorts: For invaluable feedback, discussions, and distractions.

Contents

List of Tables

List of Figures

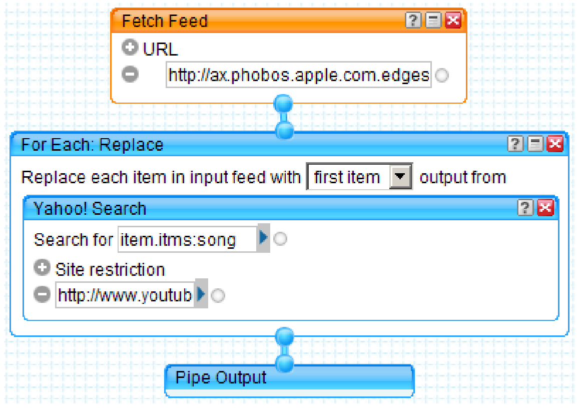
List of Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AJAX	Asynchronous JavaScript and XML
AOP	Aspect Oriented Programming
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
DSL	Domain Specific Language
DTPT	Design-Time Performance Testing
EC2	Amazon Elastic Compute Cloud
EJB	Enterprise Java Bean
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
ISO	International Organization for Standardization
J2EE	Java 2 Platform, Enterprise Edition
JDBC	Java Database Connectivity
JIT	Just-in-Time Compilation
JNI	Java Native Interface
JSP	JavaServer Pages
JVM	Java Virtual Machine
JVMTI	Java Virtual Machine Tools Interface
LRU	Least Recently Used
PDF	Portable Document Format
PHP	Hypertext Preprocessor
PNG	Portable Network Graphics
RAM	Random Access Memory
RDBMS	Relational Database Management System
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Standard Query Language
TPC	Transaction Processing Performance Council
TPTP	Eclipse Foundation's Test & Performance Tools Platform
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

Chapter 1

Introduction

Large, interconnected software systems are becoming common. Software developers can build systems that integrate with complicated applications using web services, open source libraries, and off-the-shelf components. In minutes, a web developer can create mashups that access massive data repositories including: the roadways of North America via Google Maps [21], the literature collection of Amazon [2], and one's network of friends on MySpace [22]. Figure 1.1 shows the visual source code of an example mashup integration that I recreated in minutes using Yahoo! Pipes [53] based on Nick Bradbury's example [7], to view videos of the top 10 selling music tracks on iTunes.



URL http://ax.phobos.apple.com.edges

Replace each item in input feed with first item output from

Search for item.itms:song

Site restriction http://www.youtub

Figure 1.1: An example Yahoo! Pipes mashup that integrates: iTunes sales data (top 10 songs in the US market), Yahoo! Search (search for those songs), and YouTube's video library (links to the videos for each song).

While the availability of open APIs for accessing massive data repositories empowers programmers, the abstractions that make integrations, like the Yahoo! Pipes example, so simple to create also have the potential to hide important software qualities even from those developing that software. While Yahoo! Pipes allows programmers to visualize the application data flow and processing steps, it does not illustrate how the application performs, a quality that is often important to designers of applications relying on large data sets.

While software developers have access to performance optimization tools to fix performance problems, designers rarely have access to performance information during the design process. In the case of Yahoo! Pipes shown above, this could be accomplished by executing the components of the application, collecting metrics for each components, and annotating the visualization with the metric results. In Figure 1.2 I have illustrated how these annotations might look if added to the Yahoo! Pipes visualization.

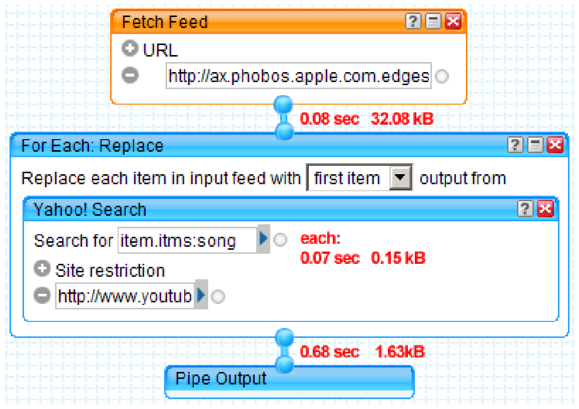


Figure 1.2: An example of how a Yahoo! Pipes Mashup could be annotated with performance metrics for fetching and parsing an XML feed.

The annotations in Figure 1.2 show that fetching the original feed takes 0.08 seconds and fetching the Yahoo! search for each item in the feed takes 0.07 seconds, and that the entire application takes 0.68 seconds to execute. To make the application run faster, a designer might suggest executing the search requests in parallel by using a parallel *for each* component. Since most of the time in executing a web request is spent waiting, executing multiple requests in parallel should give us a noticeable speed increase.

While our intuition leads us to believe that parallel requests will run faster, we should be sure that in our situation this optimization does indeed provide a real performance benefit. We could run ad-hoc tests on both versions of the application, but with the performance metric annotations introduced. Although Yahoo! Pipes does not currently offer a parallel *for each* component, if such a component was available we could simply create a second version of our Yahoo! Pipes Mashup and compare the performance annotations of both versions. Figure 1.3 shows that while executing the requests in parallel reduces the execution time of the entire application, each request takes 0.2 seconds longer (due to overhead from thread start up and competition for bandwidth and processor time).

Well designed software components, like those in Yahoo! Pipes, facilitate building large composite applications. For example, applications designed for simple, fast responses rather than complex, slow interactions are better suited to web services integration. When designing software, designers are faced with deciding between alternative solutions to a problem. Deciding amongst potential solutions prior to implementing these solutions is difficult. Although design decisions can be changed, making good decisions early is important since further design and implementation may have dependencies on early decisions.

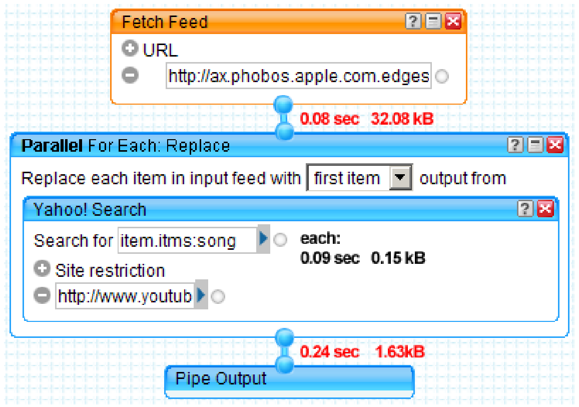

Fetch Feed
URL
http://ax.phobos.apple.com.edges
0.08 sec 32.08 kB
Parallel For Each: Replace
Replace each item in input feed with first item output from
Yahoo! Search
Search for item.itms:song
each:
0.09 sec 0.15 kB
Site restriction
http://www.youtub
0.24 sec 1.63kB
Pipe Output

Figure 1.3: The Yahoo! Pipes mashup made parallel to improve performance.

1.1 Motivation

Making software design decisions is a difficult task. Not only are design decisions made early, but many are also fixed for the duration of the project. Software designers must choose between alternative solutions to design problems and try to maximize the value of the software project with respect to a number of concerns including: cost, scope, quality, and time.

1.1.1 Software Qualities

ISO 9126 identifies six main quality characteristics that are used for measuring the quality of a software system [25]. The main qualities specified in ISO 9126 are listed with a brief description of each:

1. **Functionality** - the degree to which the specified functionalities exist and are correct in the system.
2. **Reliability** - the system's ability to respond to a given load for a specified period of time.
3. **Usability** - the effort required for the intended set of users to make use of the system.
4. **Efficiency** - the amount of computing resources used to meet user demands.
5. **Maintainability** - the effort required to fix problems and evolve the system to meet further demands.
6. **Portability** - the system's ability to be used in multiple, different environments.

1.1.2 Budget and Schedule

While software qualities are an important measure of a software system's success, the total cost and delivery time are critical to successful software projects. Software designers need to maximize

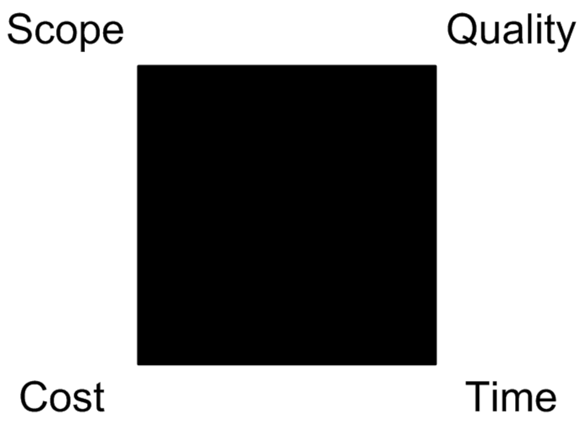


Figure 1.4: The Iron Triangle of project management, as presented by Bittner and Spence [6] in the shape of a square, shows four competing concerns in software development.

software qualities and meet the requirements of the project scope within the constraints of their budget and schedule. Balancing these four important aspects is summarized in the Iron Triangle of software project management in Figure 1.4. With six major quality characteristics, a desired system scope, budget constraints, and limited time, software designers and developers need to balance these often competing concerns. To make tough design decisions, designers should be well informed on how each alternative solution trades off the above four aspects.

1.1.3 Performance

The performance of a software system, referred to as efficiency in ISO 9126, is critical to the success of that system, yet it is difficult to predict that performance during the design stage. As I will show in related work, most performance tools are intended for use after software has been implemented, and is not intended for use during design. Thus, making early design decisions that influence system performance is a challenge during software design.

When building large applications from web services, the performance of these building blocks can determine the performance of the overall application. When composing an application from components (web services, off-the-shelf, or open source) the application may make multiple calls to the component, thereby multiplying any component performance bottlenecks in the overall application. Because of this multiplication of performance problems, performance is a critical quality in the software design of successful components and therefore applications composed of components. Since applications composed of web services and components are becoming so wide-spread, considering performance concerns during software design is becoming a requirement of many applications.

While the importance of performance compared with other qualities depends highly on the project being undertaken, all projects have at least an implicit performance requirement, and that is to provide some meaningful functionality in an acceptable time frame. Thus, all software projects need to be at least aware of major performance bottlenecks that may jeopardize the project's success. Software performance problems can cause further problems in usability, if the user feels the application is unresponsive, and correctness, if an application timeout causes the application to not return a correct response.

1.2 Thesis Statement

In my thesis I look to show that: using design test suites to collect performance data allows software designers to better estimate and compare the performance of those alternatives, leading to more informed design decisions. Software designers need to choose between alternate approaches to solving a problem, but these alternatives are not generally implemented prior to making the decision. I propose building simple prototypes of potential solutions to get a quantitative performance comparison. Using this performance comparison, designers can better compare the performance consequences of each alternative. To ensure this performance comparison accurately reflects the scenarios for which the application is being designed, designers encode scenarios as design test cases that automate the execution, comparison, and documentation of performance prototypes.

1.3 Scope

The focus of my research is to inform the process of making design decisions by giving software designers a tool to evaluate the performance consequences of design alternatives in real scenarios.

1.3.1 Design-Time

While this work could be applied to many decisions involving system performance I have chosen to focus on those design decisions that have impacts throughout the system and as such are costly to change part way through development. For decisions that are difficult to change, the cost of spending time to compare alternatives is offset by fewer design changes late in development. For decisions that can be easily changed, late-stage performance tools like profilers and optimizers can be used to achieve the same types of performance improvements.

1.3.2 Performance

While a number of important software qualities exist, my work focuses on performance. Specifically in this work when I refer to performance I mean: *the use of computing resources while running the actual software system under scenarios for which the system is being designed.* Computing resources are those limited resources consumed by a computer program including, but not limited to:

- Processing time
- Total waiting time
- Disk space used
- Memory used
- Power consumed

Scenarios are situations in which the application is designed to run and include details regarding:

- The platform and its system configuration
- The load of requests/users
- The environment in which the system is running (e.g. other applications sharing resources with the system)

1.3.3 Complex systems

Furthermore, I consider performance in the case of complex systems which I define as a system in which there is some aspect of performance that is unknown and needs to be approximated. If we are writing a program in machine language, for a processor without operation reordering, virtual memory, or pre-emption, it might be feasible to completely understand the performance consequences of design alternatives. Our ability and means to estimate performance with precision is hindered when we introduce complicated components like: high-level language optimizing compilers; operation re-ordering, multi-core processors; and multi-tasking operating systems. Furthermore, modern abstractions like virtual machines, garbage collectors, and proxies make complete, accurate analysis even more impractical; to understand systems using these abstractions we need to somehow approximate system performance.

1.3.4 Testing

Design test suites capture usage scenarios for which the application is being designed. I use these test suites to compare prototype implementations which approximate full implementations of design alternatives. Executing the prototypes against these design test suites allows designers to automatically collect approximate performance information (resource usage) of various approaches. While these approximations may not be entirely accurate, I expect to provide designers performance insight that can help them make design decisions.

For this approach to work, designers need some type of implementation with which to test. This need not be a full scale implementation; it may be a prototype built specifically for the purpose of running these tests. In the examples in this chapter I have used Yahoo! Pipes, a system for visual programming to rapidly prototype implementations.

For example, we might wish to test whether the Yahoo! Search used in Figure 1.1 should be replaced by a Google search to improve the performance. We can prototype these alternatives by simply executing a Google search and Yahoo! search for the same item and compare their running times and response data sizes. If we encode this usage situation as a design test suite, we can refine the situation, change the prototypes or build entirely new prototypes (for other search providers), to ensure our decision is based on accurate, up-to-date information.

1.3.5 Related topics

This work is influenced by and similar to other research topics, but it is important to understand how my work differs from these other topics:

Algorithm analysis is used to compare the asymptotic resource usage of solutions and summarizes this asymptotic behavior using order notation (e.g. $O(1), \theta(n), O(n^2 \log(n))$). While order notation has great value, it is an estimate of performance in the general case based on a framework for counting operations executed or memory spaces used on an abstract machine. My approach attempts to consider performance in specific cases (those of the system being designed) and allows designers to explore the constant factors hidden by order notation. In practice algorithms with slower asymptotic worst-case bounds can perform better for the set of inputs and host platform used in a specific scenario, for example: iterative vs. fast fourier matrix multiplication on small matrices, simplex algorithm vs. interior point methods for solving linear programs, and selection sort vs. merge sort for short, nearly sorted inputs.

Late-stage optimization is usually accomplished with performance profilers and code optimizers. The process and framework in my thesis are intended for use during the design phase while profilers and optimizers are intended for use on near production-ready code. Since they are not intended for use in the design phase, late-stage optimization tools are difficult to apply when comparing design alternatives. While we can discover performance problems with these tools, it can be costly to change design decisions by the time these tools can be used effectively.

Load testing tools use test cases to understand the resource usage of a system under realistic load. Although load testing and my work both use test suites to simulate realistic situations, load tests are generally used once a design alternative has been chosen and implemented to test and optimize the capacity of that implementation. My work is intended to inform the decision among multiple approaches prior to a full-scale implementation.

Performance assertions can be added to test cases to ensure they do not violate some upper bound on resource usage. Performance assertion failure indicates the program is not correct, thus performance assertions can be used to ensure correctness, but not to compare between two correct, but different solutions that may vary widely in resource usage. Qualitative performance metric comparisons of correct, but different solutions in real scenarios is the focus of my work.

Making design decisions are made by predicting and comparing the software quality attributes of potential solutions to a given design problem. My work looks to inform this process with the performance quality of potential solutions, but does not dictate how to make the design decision.

1.4 Contributions

The primary contribution of this research is a process for performance testing software designs using prototypes and test suites. This process allows software designers to capture important performance characteristics with prototypes and build test suites that simulate situations of interest. Because test suite execution, performance metric collection, and results documentation are automated, the process lets developers easily repeat tests to compare alternative implementations. This process's resemblance to conventional correctness testing should allow practitioners to employ it with more ease than current alternative approaches to performance prediction.

My work makes a number of secondary contributions as well:

- Introduction of the Design-Time Performance Test (DTPT) Framework implementation developed in Java. The DTPT Framework simplifies the process of building, evolving, and executing performance test suites while introducing only moderate performance overhead.

- I show through a set of experiments that the DTPT Framework gives results that are consistent with our intuition and established best practices.

- Further, I compared experimental results obtained running prototypes in the DTPT Framework against results obtained from a full scale implementation and showed that these results are consistent and the prototypes produce estimates that lead to appropriate design decisions in the experiments undertaken.

- I also illustrate how the process can provide guidance to software designers and developers even without framework support in a series of ad-hoc design performance tests. While other tools can be used, the DTPT Framework provides a simple way to employ the process presented in this thesis.

1.5 Thesis Overview

An overview of design-time performance testing is presented in Chapter 2. This overview first presents a taxonomy of related tools, then explores how some of these tools are used to estimate performance during software design. After considering a number of solutions I explain why I have chosen to focus on performance prototypes in this research. In a section on related work I explain the other tools in my taxonomy to explore both the benefits and criticisms of these tools.

Chapter 3 introduces my approach to the problem by introducing the DTPT process. The process is used to estimate software performance characteristics of design alternatives by building, running, and analyzing prototype test suites.

I introduce the DTPT Framework in Chapter 4 which can be used to implement the DTPT process. The framework is designed to build and test Java design prototypes. The framework automates much of the testing process and including the collection of general performance metrics.

Chapter 5 presents a number of experiments to verify that the framework produces results that are consistent with our intuition and established performance best practices. Then I show that the framework introduces only a reasonable overhead when instrumenting test cases to collect performance metrics.

The process and framework are evaluated with a real world application and potential design alternatives Chapter 6. This chapter documents experiments which show that prototypes and test suites can estimate the eventual performance of a full implementation and gives an example of performance test evolution.

Chapter 7 presents a discussion of my solution. In this chapter I consider the benefits of using test suites, the results of an informal industry survey, criticisms to my approach, out of scope issues, and potential applications for this work.

Chapter 8 presents the findings of my research focusing on the strengths and weaknesses of the proposed solution. Future work that has been inspired but not yet undertaken is explained next. The thesis ends with a final summary and review of the contributions made by this work.

Chapter 2
Design-Time Performance Testing

Design-time performance testing is the process of taking performance concerns into consideration during the software design process. By considering the performance consequences of various design alternatives, designers can foresee performance bottlenecks early and decide how to limit the impact of performance trade-offs on the software project. Design-time performance testing helps software designers avoid making early design decisions that yield poor performance in the eventual software implementation.

Performance is only one of many software quality attributes a given software designer may be attempting to maximize. The design alternative with the best performance is not necessarily the best alternative since other qualities must be considered by the designer. The importance of performance is highly dependent on the type of project (is this a device driver, or a web script?), stakeholder expectations (is this a temporary or long-term solution?), and the operating environment (is this an embedded platform with very little memory?). Even in projects where performance is less important, making an early design decision that results in significantly reduced performance can be disappointing to implementers and costly to change. Design-time performance testing seeks to inform software designers of performance consequences early so designers can make informed performance decisions.

Successful design-time performance testing relies on being able to quickly and clearly determine the performance consequences of alternative solutions to a design problem. There are two important aspects of design-time performance testing:

1. **Speed** – being able to quickly estimate the performance characteristics of design alternatives and clearly communicate the difference between alternatives early in the design process. Although designers need an understanding of the available options to choose between them, we do not want to delay the design process by fully exploring all solutions.

2. **Accuracy** – varying the level of accuracy and detail in the performance evidence to balance the completeness and timeliness of estimates. When designing software, some decisions can be made quickly with simple estimates while others require more detailed analysis. Being able to quickly evaluate some alternatives while looking more deeply at others is critical to making timely, informed decisions.

As a result of clearly the rationale for design decisions, through design performance test suites, designers can revisit and change design decisions as the project requirements change and evolve. This helps the design adapt to change while reducing the amount of rework required to gather estimates in the face of changed requirements.

2.1 Software Models

When comparing software qualities, like performance or functionality, it is often preferable to use an abstraction or simplification of the full software implementation, either because the full system is difficult to analyze or does not yet exist. A *Software Model* is a representation of a software system that may have simplifications or abstractions not present in the full implementation. Software Models allow designers to analyze a system without having to construct the full system and abstract some of the implementation details not important to the analysis.

A given Software Model exists on a continuum ranging from abstract to concrete. For the purpose of this thesis and in accordance with the terms used in related work I divide this continuum into four categories from the most abstract to the most concrete:

1. **Analytic Model** – represents a software system as a closed form mathematical model designed to explore a behaviour of interest in the software system. Tan et al. [47] present examples of analytic models for streaming video delivery.

2. **Simulation Model** – similar to an Analytic Model, but generally requiring some computer simulation to arrive at a solution (e.g. iterative solving until converging on a solution).

3. **Prototype** – an executable part of a software system that exhibits behaviour we are interested in analyzing. Prototypes may not completely or correctly implement the functionality of a full implementation. Denaro et. al [16] present example prototypes composed of enterprise application middleware.

4. **Implementation** – a complete, executable software application that performs the desired functionality of the software system.

2.2 Software Performance

In my thesis, I am interested in how performance as a software quality attribute can be estimated and considered during software design. This overview of related work looks at whether performance is important in light of increased computing power. For performance prototypes to be valuable, we need a way to gather and report performance metrics from them. The next section considers how performance metrics are gathered using profilers. I then consider how late-stage optimization tools and early stage performance estimation tools are used to improve implementation performance.

2.2.1 Is performance important?

Performance is an important concern outlined in ISO 9126 [25], but this standard was released in 1991. Since then less expensive and more powerful computing resources were developed, some might argue that performance is far less important today than it was 10 years ago.

Cost of computing

In Atkinson and Court's exploration of the new economy, they found that the cost of computing resources (e.g. processing speed, random access memory, bandwidth) has reduced significantly over the past 30 years [5]. While software applications at one time had to fit within a very limited set of computing resources, reduced computing costs have also led to an increase in the available resources to software applications. Applications no longer need to fit within 640K of memory, and even simple applications regularly exceed this historic boundary.

Since computing resources are far less scarce, designers can make decisions that sacrifice performance in exchange for other quality attributes. Sun Microsystems developed Java, a C-like language designed to run on a virtual machine providing programmers an abstraction from the underlying hardware and operating system [41]. Java exchanges some loss in performance, to present application programs with the interface of a virtual machine on a host platform, for increased portability.

Massive computing power can be purchased in inexpensive increments using systems like Amazon Elastic Compute Cloud (EC2) [3], or Sun Microsystems' Grid Compute Utility [45]. Amazon EC2 allows applications to dynamically purchase and deploy machine instances and combine these in a distributed computing environment allowing an application's environment to dynamically scale to its performance demands. Applications can rent computing power starting at $0.10 per hour per virtual machine.

While increased resource availability has given software designers far more freedom, being able to make design decisions that sacrifice performance increases designers need to understand the consequences of that tradeoff. While in the past performance concerns had to be addressed for applications to run on the environments available, today designers need a way to decide which performance concerns are relevant and which can be ignored, since using additional resources has a real cost.

Research interest

Performance problems are of active interest to practitioners and researchers. Performance implications of design decisions can be significant; Rudzki et al. found that the SOAP messaging format can introduce overhead in network communications that results in a message five times larger than the same message generated using binary serialization [39]. Cecchet et al. found that even within the framework of Enterprise Java Beans (EJB), one design, Session Beans, provided up to five times the throughput of other alternatives [11].

While Cecchet et al. and Rudzki et al. both documented the performance consequences of design decisions in n-Tier applications, understanding these performance consequences required significant up-front work. Cecchet et al. wrote nearly 65,000 lines of code [11] to compare six alternate software architectures. Software designers need a way to quickly understand performance trade-offs of various approaches to inform their decision making process.

2.2.2 Performance metrics

Profilers extract performance metrics (e.g. heap usage, time spent in each method) from running applications either by instrumenting the application with monitoring code or sampling the application periodically. Developers can use the results to find performance bottlenecks and focus their optimization efforts.

The Eclipse Foundation's Test & Performance Tools Platform (TPTP) is a framework for building monitoring/logging, testing, and performance tools [18]. TPTP includes tools for tracking memory usage, execution time, and method level profiling as well as an intuitive GUI for exploring the results of these tools [37]. TPTP also provides support for running and tracking the results of JUnit, manual, and load tests.

Javana is a framework for building performance tools that can respond to events in the Java Virtual Machine. Maebe et al. show how to write a profiler using Javana to track events like garbage collection and memory reorganization [31] using dynamic binary instrumentation. Although the overhead of Javana is a concern (applications run 90–345 times slower with Javana enabled), being able to track low-level JVM events (like heap allocation and reorganization, and method optimization) could be very important to optimizing applications running on a virtual machine.

Reducing the overhead of capturing performance data is an important concern. DTrace is a performance profiler originally developed for Solaris applications that allows programmers to dynamically instrument running binary images [10]. Cantrill et. al. explain how DTrace dynamically replaces machine instructions to create profile probes so that no overhead exists when probes are inactive. Using a domain specific language called D, performance testers can control instrumentation and filter results in both user and kernel code. Recently Apple released a tool called Xray which uses a port of DTrace to allow programmers to track events, processes, memory, file, network, and graphics metrics of running binaries in Mac OS X Leopard [4]. DTrace offers a very low overhead mechanism for collecting a number of performance metrics and Xray provides intuitive visualizations of those metrics.

Aspect-oriented programming (AOP) is a paradigm used to modularize cross-cutting concerns. Pearce et al. [35] use AOP to implement four types of profilers and consider the feasibility of profiling Java applications with AspectJ's current implementation . The aspects that implement each profiler are very succinct (under 20 lines), but their overhead is much higher than hprof, a standard Java profiler built by Sun Microsystems. Although profiling with aspects may introduce additional overhead, aspects provide an effective way to weave performance collection code into applications without directly modifying the source files of the system under test. Aspects also allow precise selection of where profiling code is to be added, allowing users to precisely target the overhead of instrumentation.

When collecting performance metrics it is important to understand the overhead introduced by the collection mechanism. TPTP, Javana, DTrace, and AOP all have interesting ways of collecting performance metrics but each has a different performance overhead. Furthermore, performance profilers are generally designed to collect perfomance metrics from a single running system and thus automatically running multiple alternative implementations through scenario based test cases and documenting the results would require some additional infrastructure.

2.3 Taxonomy of related performance tools

With the variety of performance tools available, categorizing them allows us to more easily compare each tools. I introduce a taxonomy of related tools in Table 2.1. This taxonomy compares each tool based on the important aspects identified in the overview including: stage, cost, whether performance is estimated, the level of detail provided, and if documentation is provided.

2.3.1 Late-stage optimization

Once an application has been implemented, late-stage optimization is used to make that implementation as efficient as possible. These optimizations can extract significant performance gains, but are sensitive to design, platform, and implementation changes and thus they are usually employed late in development. My work on performance prototypes does not attempt to replace late stage optimization, but rather to give designers early performance insight hopefully exposing performance bottlenecks prior to late-stage optimization, reducing rework in that stage.

Table 2.1: A taxonomy of related performance tools.

	Stage	Cost performance	Estimate of detail	Level	Documentation
Optimization	late	**low**	no	**variable**	no
Parameter tuning	late	**low**	no	low	no
Performance rules	early	**low**	**experience-based**	high	non-specific
Expert Advice	early	high	**experience-based**	high	specific
Models	early	high	**evidence-based**	high	**specific**
Prototypes	early	**low**	**evidence-based**	**variable**	**specific**

Compile-time optimization

Building software in high-level languages simplifies programming large applications, but since it abstracts details of the underlying machine some details are lost. While a compiler converts high-level language constructs to platform-specific instructions, compile-time optimizations optimize the resulting platform-specific instructions. For example, a compile-time optimizer might in-line a short, common method call to avoid the overhead of a function call.

In languages that are interpreted or compiled just-in-time (JIT), similar to those traditionally carried out at compile-time, optimization can occur while the application is running, using run-time information to guide optimization selection. Krintz explored two dynamic optimizations implemented by modern Java JIT compilers: hot method optimization, in which frequently used methods are optimized further than other methods in the program; and hot call-site in-lining, when frequently executed call-sites in-line the method they are calling to reduce method call overhead [29]. Krintz implemented these optimizations using IBM's JikesRVM, and applied them based on static and dynamic performance information. Krintz's resulting implementation reduced total execution time of Java applications by 75% compared with an unoptimized application performance, and improved upon performance gains by traditional static optimizers.

Optimizations can be beneficial in some situations and harmful in others; knowing when to apply optimizations is a difficult problem, but since optimizers in general give good performance, this problem is sometimes overlooked. Zhao et al. explored how to decide which optimizations to apply using an analytic model to predict performance improvements of two specific compile-time optimizations [55]. The authors' framework makes accurate predictions in 80% of tested cases, leading to performance improvements from 8-10% over unguided optimization.

Krintz's work demonstrates that collecting run-time performance information is critical to making performance decisions, and Zhao et al. showed how accurate performance estimates can result in improved implementation performance.

Parameter tuning

Complex software often has a number of tunable parameters that influence the application's performance. By tuning these parameters, users can optimize the application for a given scenario. Database systems often have a number of parameters including cache size, cache behaviour, connection pool size, index format, and data layout format. This allows a single database application to service the needs of various applications with disparate data access requirements by simply tuning these parameters for the desired usage scenario.

Performance tuning of runtime parameters (e.g., heap size, caching algorithm, thread pool size) can yield major performance benefits. Dalal et al. introduced a methodology of exploring the universe of potential parameter values to yield significantly better results than random searching [15]. Sullivan implemented an automated system for exploring parameter values for four parameters of the Berkeley DB system [43]. Sullivan used influence diagrams to guide probabilistic reasoning and decision making to tune an actual system under a number of different workloads and in 90% of cases found configurations within 10% of optimal.

Performance tuning relies on the fact that performance of a software system depends on the scenario under which it is used. Delal et al. and Sullivan both found ways of tuning implementation parameters to optimize those implementations for specific situations. While parameter tuning allows users to optimize performance for a specific usage scenario, developing a parameter tunable application is significantly more complicated, and testing that application is even more complicated since one must be sure parameter combinations do not change the correctness of the application.

2.4 Early stage testing

There are a number of approaches to estimating software performance at design-time. In this section I consider four general approaches: following general performance rules, seeking the advice of performance experts, employing simulation or analytic models, and building a prototype for performance testing.

2.4.1 Performance Rules

Software developers have come up with a number of general guidelines for building efficient applications based partly on their experience and partly on the underlying data structures and algorithms. For example: when deciding whether to use a hash table, or binary tree to store a collection of objects, developers may consider a table of operation timings (Table 2.2). Many developers have this table committed to memory so the decision between a hash table and binary tree is nearly instant. Furthermore, developers may recall the data structure used in a similar situation and simply use the same structure.

Table 2.2: Average time to complete dictionary operations and iteration when using a hash table or a balanced binary tree [13].

Operation	Hash table	Balanced Binary Tree
Insert	$O(1)$	$O(\log_2 n)$
Delete	$O(1)$	$O(\log_2 n)$
Search	$O(1)$	$O(\log_2 n)$
Sorted Iteration	$O(n \log_2 n)$	$O(n)$

The Eclipse Foundation identified thirteen techniques that can be used to avoid common mistakes made by developers when building plug-ins for the Eclipse platform that lead to poor performance [36]. These techniques range from saving memory when extracting substrings to reducing startup overhead and memory consumption by using lazy activation. By identifying common performance mistakes and showing how to resolve them, the Eclipse Foundation hopes plug-in developers can design and build plug-ins that yield better performance.

While some of the performance bloopers identified by the Eclipse Foundation are directly related to Eclipse plug-in development, most of the performance recommendations are good recommendations for any Java programmer. By learning about these performance bloopers, software developers can prevent some common performance problems from ever being part of their application. Performance rules can be observed in early stages of development, which reduces the amount of time spent redeveloping and optimizing code that is performing poorly late in the development cycle.

Performance rules are usually general principles and practices. The estimates provided by performance rules are based on the experiences of others and while some of these experiences are thoroughly documented, the documentation provided has a broad audience and thus is not specific to the project a given developer is working on.

Another list of performance rules can be found at O'Hanley's website [33]. O'Hanley keeps a current list of Java best practices on his website, many of which are recommended based on their performance characteristics. O'Hanley explains and provides source code for the recommended practice and in many cases, a comparison of the performance between the recommended practice and alternatives. O'Hanley's website explains both why developers should follow these practices and what kind of benefit to expect.

In an article on J2EE applications, Ramachandran discusses a number of design patterns that try to eliminate performance bottlenecks common in J2EE applications [38]. These patterns help designers address performance considerations that are unique to n-Tiered applications and are relatively high-level.

The performance rules prescribed by O'Hanley, the Eclipse Foundation, and Ramachandran can be useful to designers if the decisions they are considering are covered by these articles. The articles explore common performance problems and are available for free. Unfortunately, these articles cover only a small subset of the design decisions software designers face; since applications may each have very unique design decisions, finding an article that covers a specific decision may be difficult. Furthermore, most of the issues considered by O'Hanley and the Eclipse Foundation are low-level implementation details which are targeted more to software developers than designers. For those higher level rules, most are focused on just a single component in isolation rather than on characteristics of interactions between components. Finally, for many of the rules stated the authors rely on their experiences and ad-hoc testing methodologies giving readers no way to reproduce their results, or modify the scenario to fit the reader's situation.

Even general performance rules may be limited in their ability to predict performance in a specific scenario. Since the resources available can vary significantly across platforms (consider power available on an embedded device vs. a workstation), performance rules may not yield the same results on each platform.

2.4.2 Expert Advice

When making a design decision that must be fixed early and is critical to the performance of the resulting system, software designers may choose to seek advice from a performance expert. Performance experts have experience optimizing performance for applications similar to the system being designed. By drawing on their experience, performance experts can help designers understand the consequences of various alternatives.

Once an application has been implemented, performance experts use profiling and monitoring tools to optimize the application's performance. With their profiling, monitoring, and optimizing experience, performance experts can make predictions regarding the performance of design decisions [40, 52]. With the predictions of experts, software designers can make more informed performance decisions.

Software designers can involve performance experts very early in development, to avoid performance problems inherent in the system design and its implementation. Performance experts with the skills required to make accurate recommendations and a deep understanding of the problem domain can be difficult to find and costly to employ. Performance experts can provide project specific advice at many levels of detail.

Performance experts are not an option for every software design project. Projects may not have the internal experience required or the budget and time required to bring in an external expert and some applications may be unique such that few if any performance experts are available. Furthermore, since any changes may require the performance expert to reconsider their recommendation, this approach is difficult to apply since requirements often shift during the development of a given software project.

2.4.3 Analytic and Simulation Models

Analytic models describe key components of an application while abstracting those details that are not important. Often these models consist of components sending messages through queuing networks, and can be employed early in development to understand the performance consequences of a given system architecture [49]. Simulation models seek to simulate the components in a system as well as the workload those components are expected to handle. Using models allows designers to clearly understand how different parameters values and component configurations affect application performance [47].

Tan et al. [47] used an analytic model to determine server performance metrics such as mean client waiting time, and client drop rate of hierarchical stream merging, a technique in on-demand content delivery, versus traditional delivery. The analytic model developed allowed designers to consider the performance consequences of this technique under a number of different client loads. Furthermore, since the modeled system consists of a massive content delivery network, implementing the system to determine performance gains would be very costly.

Jin et al. showed how analytic models can be used to predict the scalability of legacy information systems [27]. The authors present an approach that uses benchmarks and monitoring of the existing system to parameterize and validate an analytic model of the system that can then be used to explore future workload changes. This technique is used to understand the performance benefits of a software architecture change to an existing production system (monitoring customer electricity usage for a utility company). Jin et al. found their model was accurate (within 8% of benchmarks), and helped designers choose the new system architecture.

Models can be used early in the development cycle to understand the software performance at various levels of detail. While these models can provide valuable insight into a system's performance, models often require parameterization (e.g. arrival rate, seek time, service time distribution) based on empirical analysis. Denaro et al. found that models can be very expensive for real systems [16], but models can often be reused within a given project and across multiple similar projects. While models can be designed for any level of detail, a given model is designed to understand a single level of detail. Once a system model is working, it can be used to explore a range of parameter values, but generally does not provide documentation to explain why certain parameter values produce a given output value.

Simulation and analytic modelers need a number of specialized skills to build accurate models (e.g. an understanding of statistical distributions, familiarity with queuing networks). Furthermore, since models may not be easily adapted in the face of design or requirements changes, it seems that while some design decisions warrant the use of modeling techniques, they are not practical in the general case.

2.4.4 Solution Prototypes

In the same vein as building simulation models, designers can prototype the important parts of design alternatives and compare the performance of those prototypes. While prototypes will not replicate the exact performance of a full scale implementation, a well constructed set of prototypes can provide accurate estimates that can be used to compare alternative approaches.

Distributed systems are often written using frameworks like J2EE and CORBA. Denaro et al. found that while these frameworks provide facilities for communication, transactions, and security, it is these services that determine the performance of most distributed systems [16]. Since framework performance is critical to designing efficient distributed systems, Denaro et al. explored using application specific prototypes that compose middleware functions to predict system performance. The authors validated this approach by showing that their performance prototypes, built using existing middleware, accurately model the performance of a full scale distributed system implementation based on that middleware.

When designing large distributed systems, designers are presented with a number of decisions regarding the system's architecture. Krüger et al. explore how prototypes built using aspects can be employed to predict performance consequences of service-oriented architectures (SOA). In their work, Krüger et al. built prototypes of the various services in the distributed system, and weaving these together using aspects. In this way, designers can use a single set of prototypes to compare the performance of multiple alternate architectures. Furthermore, performance measurements are captured using aspects allowing designers to easily weave measurement code into each alternate architecture.

Prototypes approximate specific parts of a potential system. By running these prototypes with a benchmarking tool, software developers can obtain evidence-based performance estimates early in the software development cycle. Prototypes can be constructed to approximate different parts of a system at any desired level of detail. The major cost of this approach is building, executing, and benchmarking prototypes on various performance metrics. If designers can approximate performance with a reasonably simple prototype, the cost of prototyping can be quite low.

Prototyping design decisions to gauge the performance consequences of various alternatives seems to work well in the situations explored by Krüger et al. and Denaro et al., but can these techniques be used in other contexts? While middleware bootstraps the prototyping process, is it possible to use prototypes and performance test cases when middleware is not the key determinant of software performance and can we consider design decisions not involving middleware? Although aspects seem to allow designers to quickly prototype different compositions of components, there may be a general approach to prototyping that does not rely on having middleware and implemented components.

2.5 Documenting Performance Design

Ad hoc performance metrics can be difficult to understand, hard to reproduce, and ill-suited for comparison. Comparing design alternatives and recording rationale with only this type of performance information would be difficult. In this section of related work I review automated software documentation and how design decisions are informed.

2.5.1 Documenting software

Software documentation is an important product of software development. Software designers need to produce documentation that explains the system design for other designers, implementation developers, and system maintainers. Furthermore, documentation that explains the rationale for selecting a given design alternative could be useful when making further design decisions on the same project, re-evaluating design choices, and informing future software systems.

In this section, I consider two bodies of work related to efficiently producing up-to-date performance documentation. First, I consider how automated documentation systems ensure documentation is up-to-date; and second, I look at how test cases can document run-time system properties like correctness and performance.

Generating documentation

Developers and maintainers rely on software documentation to understand the system design and overall architecture, but documentation is often out of date [9]. While documenting the Java API for public use, Kramer found that keeping documentation accurate and up-to-date was not only essential to the project's success, but also difficult [28]. This led to the development of JavaDoc, an automated system for generating API usage documentation based on the Java source code and special annotation comments within it. Kramer believed developers were far more likely to keep documentation in the source code up to date than in external manuals, so he used this internal documentation to generate the external documentation needed by the Java API's end users.

Doxygen is similar to JavaDoc, but was originally developed for documenting applications written in C and now supports a number of languages. In addition to documenting API usage information like JavaDoc, Doxygen generates useful graphs for exploring file reference and class inheritance [50]. Doxygen supports exporting documentation to a number of formats including HTML, Latex, and PDF.

Generating documentation can save software projects time and money by reducing documentation maintenance and reducing rework caused by inaccuracies in out-of-date documentation. In a study of industry documentation techniques, Forward found that automated tools like Javadoc and Doxygen are frequently used on real software projects to produce documentation [20].

Janicki and Parnas used a tabular representation to clearly show the specification of a program in relation to the value of its parameters [26]. Cunningham and Shore developed a Framework for Integrated Test (Fit), that is designed to show requirements, test cases, and results in a colour coded table [14]. Janicki and Parnas provide a method of succinctly specifying application requirements while Cunningham and Shore have applied this to showing test results.

Having accurate documentation is important to facilitate the comparison of design alternatives. While automated tools keep documentation synchronized with the implementation, the generated documentation focuses mainly on the details of implementation (method signatures, inheritance diagrams) rather than design decisions, and does not incorporate dynamic data including performance results. Janicki and Cunningham present interesting methods for presenting design and run-time data, but provide no means to generate these types of documentation.

In my work I use test cases to automatically extract data regarding usage scenarios and automatically generate documentation like that of Janicki and Cunnigham.

Test cases

Test cases are generally used to facilitate automated testing of important usage situations. Unit tests allows developers to gauge the correctness of their implementation and performance assertions can be used to institute upper bounds on resource usage. I am interested in how test cases serve to document the scenarios they test, and how we can automatically gather run-time data by executing test cases.

JUnit is a popular testing framework that is used in industry and academia to write automated unit test cases for Java applications [34]. JUnit makes test cases easy to write, automates their execution, and clearly presents the test case results to the end user. Although JUnit is not designed for performance testing, it is commonly used for correctness testing.

Lencevicius et al. wrote a framework for implementing multi-threaded performance assertions for mobile devices [30]. Performance assertions are constraints on the running time of applications that can be tested automatically. Performance assertions, like correctness assertions, pass or fail and thus they require the test-case writer to specify some upper bound on some resource usage (usually running time) which may require intricate modeling of the operation since the upper bound may be based on a number of parameters.

Slosarski's nTime [42] is a framework similar to JUnit that executes performance tests written for the .NET framework. nTime can run multi-threaded application tests and enforce static upper bounds on a number of time metrics including total running time, % processor time, and executions per second. Zhang's p-Unit [54] is a performance unit testing framework for Java applications similar to nTime. p-Unit supports parameterized test cases which allow test case writers to specify a number of different test configurations with a single test cases, and visualize how those parameters influence performance graphically.

While correctness test cases and performance assertions provide valuable, automated feedback, pass or fail testing does not provide the right level of granularity for all situations. Furthermore, establishing upper bounds for performance assertions can be difficult to maintain and depend greatly on the execution platform. Both unit testing and performance assertions use test cases to encode usage scenarios and automatically document the run-time results of those scenarios for application testers. While pUnit and nTime are designed to test that a single system does not exceed a given performance upper bound and report if it does, my work looks to facilitate the comparison of performance estimates for various alternatives and present that comparison in a meaningful way.

2.5.2 Making design decisions

Design decisions are made early, constrain the final implementation, and are difficult to change; thus, understanding the value of alternatives is important. Sullivan et al. used Baldwin and Clark's design structure matrices and net options value to compare alternative modularizations [44]. In their analysis, the authors attempt to compare the real value of different modularization options by assigning values based on the cost of experimenting and replacing modules within the system. The authors state that "such models need not be perfect. However, models must capture the most important terms and their assumptions and operation must be known and understood so that analysts understand and can evaluate their predictions." [44]

2.6 My approach

Although each of the early-stage approaches provides insight during software design, solution prototypes seem to allow designers to quickly compare alternate design decisions at least in the situations considered by Denaro and Krüger. Furthermore, Denaro et al. showed that these prototypes can be rather accurate (within 9.3%) when compared with a full scale implementation [16]. Thus, I decided to expand on the work of Denaro and Krüger by exploring how prototypes can be built to explore performance of systems in the general-case, not requiring middleware, aspects, or implemented components. Furthermore, I explore how a process can be used to execute test cases, capturing performance metrics, and generating performance documentation.

While performance rules and expert advice may provide intuition more quickly, the level of performance detail under consideration is easier to adjust with performance prototypes. We can build simple prototypes initially then refine and evolve as necessary to produce good estimates of system performance under different design alternatives. While models are by definition abstractions of the solution details, building multiple models to handle different levels of detail would be far more costly than evolving a simple prototype implementation.

Finally, by weaving measurement code into prototypes automatically, Krüger showed how prototypes can make collecting comparable performance results easier. Furthermore, the prototypes themselves are documentation of the assumptions and abstractions employed to create the performance predictions. Software designers can examine the prototypes to prevent mistakes and/or miscalculations.

In Sanfarasetty et al., a group of industry performance experts recommend iterative, early testing with specialized performance testing tools and avoiding late "house on fire" tuning [40]. Solution prototypes designed to predict implementation performance of various design alternatives seems an good way to inform performance decisions early in development and avoid major performance problems late in development.

While many of the existing tools like hprof or p-unit could be used to collect performance results from prototypes, these tools are designed for late stage optimization and thus do not have many of the capabilities I was hoping to have in my solution including:

- Automated execution of multiple implementations through a given set of tests
- Clear definition of the tests being executed and performance metrics collected
- Automated documentation of results

In this thesis I propose a process and framework to facilitate early understanding of performance qualities early in the software development process. This work is based on the work of Denaro and Krüger, but extends their architecture specific approaches, middle-ware and service oriented architectures respectively, to general purpose prototyping. I also provide further validation that prototyping produces estimates consistent with full implementations.

2.7 Summary

In exploring related work, I have found that despite increasing computing power, performance is still an important software quality. There are a number of methods for gathering performance metrics with varying levels of overhead. Many of these tools have been developed for use in late stage optimization of software systems through compiler optimization and parameter tuning. Some types of software documentation can be extracted directly from source code and special comments in that code; furthermore, test cases can be used to produce documentation of a system's run-time behavior in a specific usage situation. Finally, I looked at how models that are not entirely accurate, but provide good predictors, can be successfully used to compare design decisions.

I also introduced a taxonomy of tools for understanding performance qualities and explored each tool including a number designed for comparing the performance of design alternatives early in the development cycle. For my approach, I have chosen to use performance prototypes since designers can: develop prototypes quickly at low cost, vary the level of detail in a prototype easily, and record the performance results of a prototype automatically.

In the next chapter, I will present a process for design-time performance testing based on building, executing, monitoring, and documenting performance prototypes.

Chapter 3
DTPT Process

In this chapter, I present the Design-Time Performance Testing (DTPT) Process that outlines how to predict performance consequences of design decisions using design test suites. I start by introducing the main goals of this process, then give an overview of the approach. I introduce a high-level idea of how we would like to capture actual performance and adapt that to estimate performance trade-offs, then highlight the benefits of using the DTPT process. I introduce two possible approaches to implementing this process before concluding the chapter.

3.1 Terminology

Fidelity: - design, analytic model, simulation, prototype, implementation

3.2 Goals

The process I have developed seeks to accomplish a number of important goals identified in related work on predicting performance in the previous chapter. These goals are outlined below:

- **Simulate real scenarios** in which the alternatives will be involved. While performance rules and experts can provide insight into general-case performance, my approach focuses on the actual scenarios of a given software system. This focus exposes special cases and constant factors that may be relevant.

- **Quickly prototype design alternatives.** Being able to estimate performance early allows designers to make informed performance decisions. Keeping these estimates light-weight ensures designers are not delayed by the overhead of comparing alternatives and also facilitates adapting prototypes when requirements change or evolve.

- **Extract relevant performance metrics.** Since this approach is interested in the scenarios specific to a given software system, it is also important to have the flexibility to focus on those performance metrics important to the design decisions in question.

- **Compare quantitative performance metric estimates.** While pass/fail indicators have had success in correctness testing, making design decisions involves trade-offs. Quantitative performance metrics allow designers to easily compare alternative implementations.

- **Explore promising solutions in more detail if needed.** Varying the level of detail explored is important so poor solutions can be eliminated early without investing too much time, while still allowing designers to explore promising solutions in as much detail as needed.

3.3 Overview

I have developed a process that allows designers to estimate the performance consequences of design decisions. While not all design decisions require performance comparisons, when a designer believes a decision between alternatives may involve important performance trade-offs, the designer can use this process to explore those trade-offs.

The important features of this process are as follows:

1. **Designers formalize scenarios as design test suites.** These design test suites setup and specify the scenarios important to making a design decision. Test suites are represented as Java classes similar to the industry standard way jUnit and nUnit test cases are specified.

2. **Design test suites are run against prototypes of each alternative.** Designers can quickly implement prototypes intended to capture only the important performance details of each alternative. These prototypes may use the intended data structures/algorithms, but need not be complete, robust, or entirely correct. It is more important for this process that the prototype approximates the runtime performance characteristics of a full implementation.

3. **Design test suites specify those performance metrics to be collected.** Designers can select exactly what performance metrics are to be collected so as to focus their investigation only on metrics that are important to the scenarios under test.

4. **Running design test suites produces comparable, quantitative results.** When design test suites are executed, the values of performance metrics can be summarized for easy qualitative comparison.

5. **Prototypes and test suites can be evolved over time.** To allow designers to explore alternatives in varying levels of detail, cope with requirements changes/evolution, and capture new usage scenarios, the process allows prototypes and test suites to evolve to meet the needs of designers. Ideally, test suites and prototypes can be stored in a source code repository to support versioning of test cases and prototypes.

Figure 3.1 shows and describes the phases in the DTPT process. One important feature of this process is that all approximation from real scenarios and implementations occurs in the first step when building prototypes and design test cases. Furthermore, these approximations are formalized in the prototypes and test cases so they can be reviewed, revised, and evolved as needed without looking beyond the first step in the process.

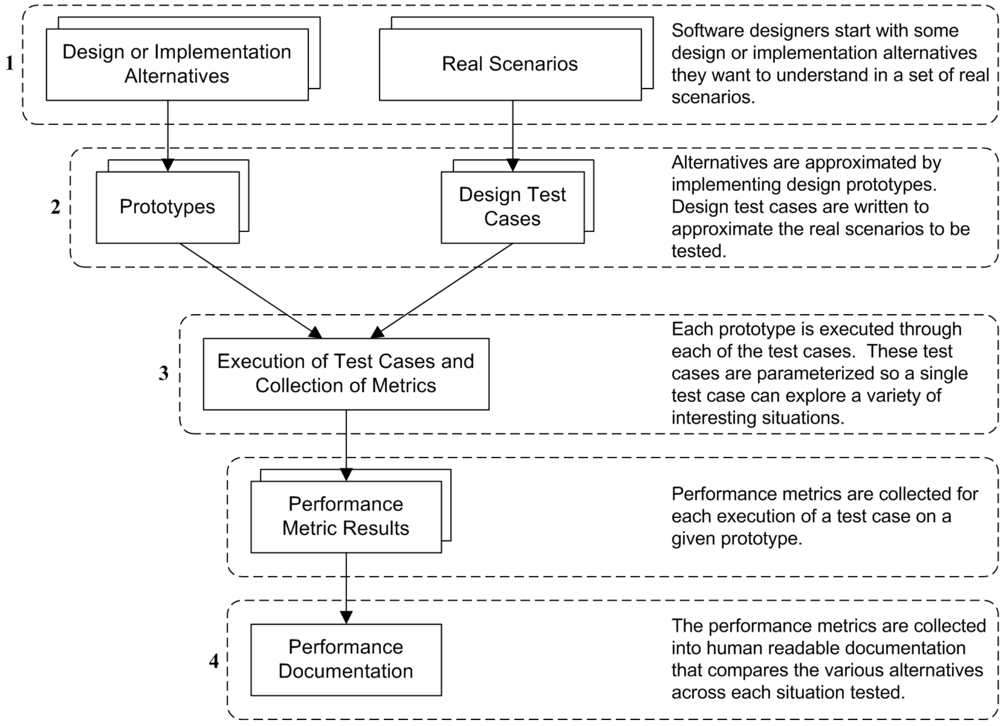

1
Design or Implementation Alternatives
Real Scenarios
Software designers start with some design or implementation alternatives they want to understand in a set of real scenarios.
2
Prototypes
Design Test Cases
Alternatives are approximated by implementing design prototypes. Design test cases are written to approximate the real scenarios to be tested.
3
Execution of Test Cases and Collection of Metrics
Each prototype is executed through each of the test cases. These test cases are parameterized so a single test case can explore a variety of interesting situations.
Performance Metric Results
Performance metrics are collected for each execution of a test case on a given prototype.
4
Performance Documentation
The performance metrics are collected into human readable documentation that compares the various alternatives across each situation tested.

Figure 3.1: The DTPT process helps designers move from a set of design or implementation alternatives and a set of real scenarios, to documentation comparing each alternative on various performance metrics.

Software designers can inform and enrich their design process by using this process for design-time performance testing. When designers need to make decisions between alternatives and are interested in the performance qualities of those alternatives, they can use the process to inform their decisions. Below I illustrate how a software designer or developer might apply the DTPT process with the example given in Chapter 1, and illustrated in Figures 1.1 and 1.3, regarding using a parallel or sequential *foreach* construct when fetching data from a web service.

1. Define Requirements: Designers define what needs to be accomplished and begin exploring alternative solutions. Designers should also think about situations in which the alternatives will be expected to operate. For example, we want an application that retrieves a set of search results based on a source XML feed (the iTunes top sellers list). We expect this application will be deployed on our web server and service 1,000 users.

2. Prototype Alternatives: the designer builds a simple prototype of both approaches. Figures

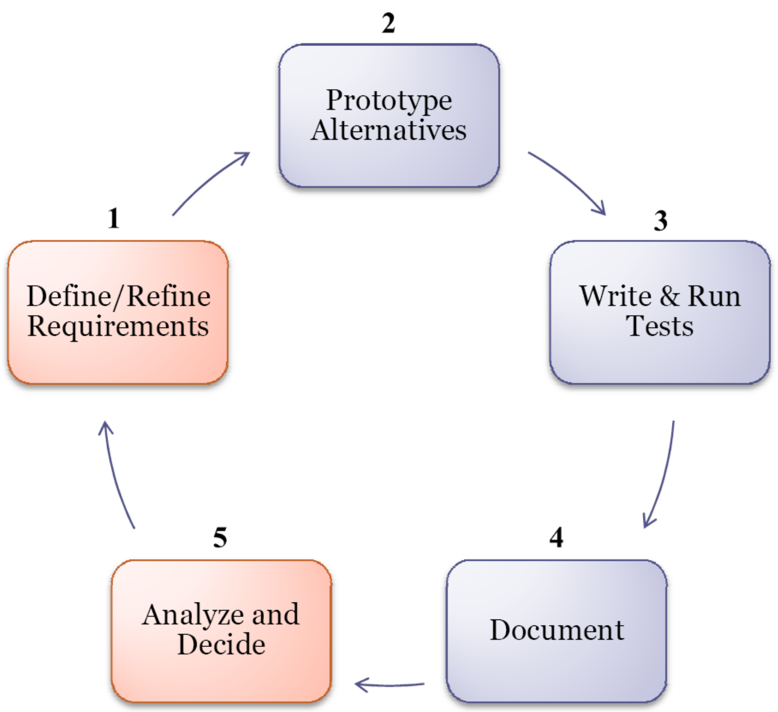


Figure 3.2: This diagram illustrates how the output of the process can be used to fine tune the process (in an iterative fashion).

1.2 and 1.3 illustrate how a designer might use a prototyping tool like Yahoo! Pipes to construct working prototypes in minutes. This stage also involves approximating real scenarios by writing design test cases which can be run against the prototypes to test important scenarios. In our example, we might be interested in optimizing the number of users this application can service. We can write a test case that issues local requests to the application; a given request rate can be used as an approximation for some volume of users accessing the web service.

3. Run Tests: We then run the test cases against each prototype and collect performance metrics during exeuction. For the example we might be interesting in the CPU load, and memory usage metrics during execution.

4. Document: once the test cases have been run against the prototypes, designers will want to have documentation that summarizes those results. For our web service comparison, we might wish to construct a table like Table 3.1 to summarize how each approach performs in the scenarios tested.

5. Analyze & Decide: documentation produced by the process summarizes the performance differences between alternatives. Designers can then consider performance along with other

Table 3.1: Comparison of the performance of three alternate web services when issuing seven requests sequentially or in parallel.

	Sequential		Parallel		Difference	
	Memory (kb)	Throughput (requests/sec)	Memory (kb)	Throughput (requests/sec)	Memory	Throughput
Amazon	1,090	1.15	2,341	5.27	114%	358%
Ebay	1,212	0.42	2,493	1.67	106%	298%
Yahoo!	2,924	0.22	2,873	0.68	-2%	209%

qualities to make a decision. In the case of our example, a parallel solution might be far more costly to build and test than a sequential solution (deadlocks, simultaneous connection limit) and for only a minor improvement in running time, it may not be worth the additional effort.

6. Refine Requirements: if in the previous step a decision cannot be made, or if new information changes the application requirements, designers can refine the project requirements. If designers feel these changes may influence performance, prototypes and/or test cases can be modified to reflect these changes, and the test cases re-run. In the example, if the source XML feed is updated predictably, for example at 11am daily, we can cache the feed without losing accuracy. This change means requesting the source XML feed will be virtually instantaneous, thus speeding up the secondary requests may give us a far larger percentage speed up than when the service was requesting the source on every execution.

3.4 Model Inputs

The process also identifies a number of inputs required to generate performance metric results as shown in Figure 3.3. These inputs are important to consider when making performance comparisons.

3.4.1 Prototypes

Instead of implementing each design alternative, **prototypes** are used to approximate the run-time behaviour of actual implementations. Because a prototype is a simplification of a full impelementation, a prototype can generally be created much quicker than the full scale implementation it approximates.

Prototypes are written as software applications that can be executed to obtain performance results. When building prototypes, it is important to capture the performance behaviour of the design alternative being approximated, while removing details that do not significantly influence

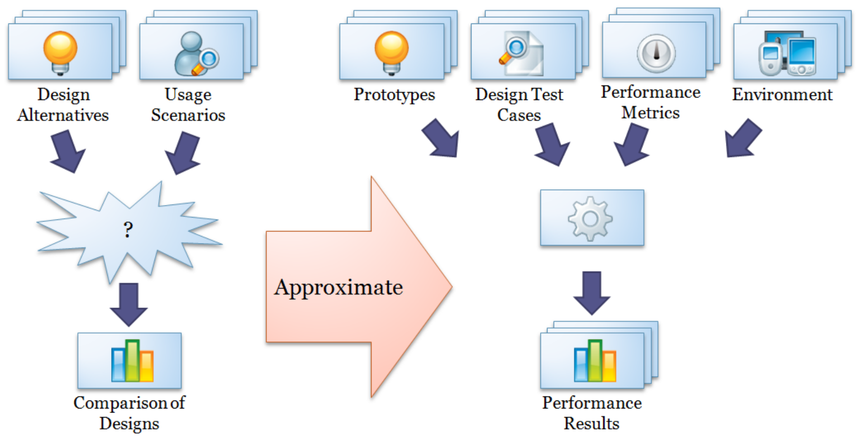


Figure 3.3: The DTPT process approximates comparing design alternatives in usage scenarios. Here we show that approximation and identify additional parameters that are important in the DTPT process.

performance behaviour.

Prototypes capture approximations in as executable code. While comments in the code could be used to capture the intent and assumptions present in a prototype, there is not guarantee that such comments will be accurate, or remain up to date.

3.4.2 Design Test Suites

In step one of the process, designers identify important scenarios the system will need to operate under. Designers might document these as Use Cases or Technical Memos in the Rational Unified Process model[1].

A design test suite is a piece of executable code that can be run to simulate a given scenario. Design test suites are used to capture the details of the scenario each prototype will be tested under. The development of design test suites is driven by the requirements identified by designers, thus they can be built based on the Use Cases and Technical Memos developed by system designers. By following this process, the intent of each test suite is captured by the Use Case or Technical Memo it is based on.

3.4.3 Performance Metrics

Performance metrics are collected during the execution of a design test suite. Designers specify those performance metrics that, given the scenario being tested, will help them decide between the alternatives.

[1]For more information on the Rational Unified Process, see: http://en.wikipedia.org/wiki/IBM_Rational_Unified_Process

3.4.4 Environment

The environment describes the state of the system's surroundings that may influence its behaviour. While the details of an environment could be unique to a given design decision, some common elements include details of the host platform, other processes running on the host, and battery power level (for mobile platforms).

Capturing the environment is important since even a minor change in the environment may signficantly influence the system's overall performance.

3.5 Anticipated Benefits of the DTPT Process

In formulating a solution process as presented above, I hope to provide a number of important benefits to users of the DTPT process:

The process introduced uses a function that maps usage scenarios and design alternatives to a comparison of the designs. This is accomplished by approximation. Because the process uses a function, repeating an experiment with the same input parameters will give the same result consistently. In the case that designers need to revisit a design, they know that the performance estimates for a given set of input parameters will not have changed. In this way, designers can avoid re-running experiments and rely on the quantitative results obtained in earlier tests.

Since the prototypes and test cases are executeable approximations of implementations and scenarios, the process records the approximations involved in using this approach within the source code of these approximations. While the results using these approximations will not be exactly the same as those produced by a executing full-scale implementations in real scenarios, being able to obtain an estimate of performance metrics early provides insight to many design decisions where obtaining complete information is not possible.

Furthermore, design test suites and prototypes are artifacts that express any simplifications or omissions from a complete implementation. If designers are suspicious of the results obtained, or want to better understand a given design alternative, they can directly examine the prototype of that alternative and the design test suite against which it was run to find any flaws or understand its assumptions.

By identifying the individual inputs to the process, software designers can easily compare performance by changing only a single input at a time. In this way, performance changes can be attributed to the change in the manipulated input. For example, when comparing the performance of alternative implementations, designers should be careful not to alter the scenario, performance metrics collected, or environment across different prototypes.

Using design test cases to approximate scenarios and prototypes to approximate alternative implementations allows designers to vary the level of detail present in either approximation when needed. Designers can run a quick, simple prototype to gather general information and rule out obviously poor solutions, then increase the detail in promising solutions to find specific performance trade-offs without having to expend this effort for less promising solutions.

3.6 Process Implementations

The process presented requires some function that can map the identified inputs to a set of performance results and is sufficiently general to support multiple implementations of such a function. In the next chapter, I describe a framework written in Java for writing, executing, and documenting design test suites to estimate performance. In some situations during the course of my thesis work I have also found it useful to implement ad hoc test suites outside the framework. Both approaches fit within the process described; in Chapters 5 and 6, I present example test suites using both approaches.

3.7 Conclusion

In this chapter, I presented a process for design-time performance testing. The process seeks to achieve a number of goals, and has a feature set to addresses these goals. I have given an overview of the process, explored its phases, and demostrated how it can be used in an example situation. By recording approximations in the program code of prototypes and test suites, the process records explicitly what approximations are being made. After developing and explaining the process I discussed a number of benefits that it provides. Two possible implementations that are introduced in the following chapters.

Chapter 4
DTPT Framework implementation

In this chapter, I explain the DTPT Framework implementation of the process introduced in Chapter 3. The DTPT Framework is intended to help designers write, execute, and document design test suites. I first present an overview of the framework including a class diagram, then explain how the major components collaborate followed by an look at the important functionalities provided by the framework. I explain the role of test cases in the framework, and then consider both criticisms of the implementation and further ideas before presenting an example usage scenario to conclude the framework introduction.

4.1 Overview

The Design-Time Performance Testing (DTPT) framework is intended to execute design test cases and prototypes written in Java. The process of writing, executing, documenting, and refining test cases in the DTPT Framework follows the process outlined in Chapter 3. To introduce the framework, I begin by outlining the framework's usage with a UML communication diagram and explanation.

4.1.1 Core Components

There are a number of core components in the DTPT Framework and each are explained below:

Prototype – a Java implementation of some design alternative. The prototype is usually a simple implementation and may not be entirely complete so long as it approximates the performance behaviour of the design alternative.

Test Case – a Java class that extends the framework *TestCase* class and will be used to execute some test against a prototype. Implements a *run* method that is called with the prototype and test case parameters as arguments.

Test Run – a Java class provided by the framework that stores the test case parameters and metrics for a single *TestCase* instance. This class will iterate through all combinations of parameters and for each it will: instantiate the desired metrics, start the profiler, execute the test case against each prototype, then record and store the results.

Test Suite – a Java class provided by the framework which contains a set of *TestRun* objects that will be executed sequentially.

Driver Class – an executable application written by the user that creates and executes a *TestSuite* by instantiating *TestRuns* and adding them to the *TestSuite*.

4.2 Communication Diagram

Figure 4.1 illustrates how the various classes within the framework communicate. Each step of the communication diagram is explained below:

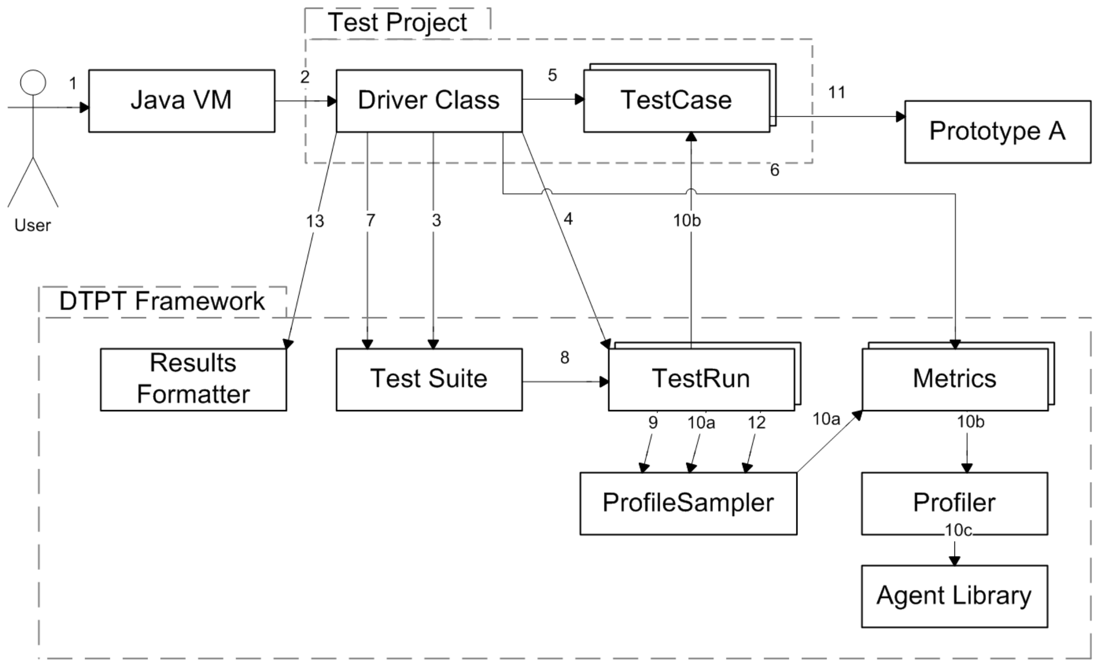


Figure 4.1: Shows how various classes communicate to execute a test case and collect performance metrics.

1. User starts the Java virtual machine from the command line.

2. The virtual machine calls the *static void main(String[] args)* method of the driver class.

3. The driver class creates a *TestSuite* instance.

4. The driver class adds a number of *TestRun* instances to the test suite.

5. Each test run is associated with a single test case that it will execute. The case parameters and run parameter matrix are also added to the test run.

6. The driver class adds the metrics to be collected to the test run.

7. The driver class calls the *execute()* method of the test suite.

8. The test suite calls the *execute()* method of each test run.

9. The test run calls the Profile Sampler's *startSampling()* method and adds each metric to the profile sampler. This starts a thread to collect the sampled metrics.

10. The test run calls *startCollecting()* on the profile sampler to begin metric collection. The profile sampler will then make calls to the individual metrics being collected (10a), which then call the profiler to collect metrics (10b) using the agent library (10c) periodically until collection is stopped. The test run also starts execution of the test case by calling *run(object[] runParams)*.

11. The test case will begin execution of the prototype.

12. Once the test case has finished executing, the test run stops the collection of metrics by the calling the profile sampler's *stopCollecting()* method. Steps 10-12 are repeated for each combination of run parameters in the matrix of run parameters. After the final run, the profile sampler thread is stopped by calling *stopSampling()*.

Some steps regarding the setup and finalization of the test case are not included in the collaboration diagram so as to simplify the diagram. Just before starting the sampling thread, the *setupCase(object[] caseParams)* method of the test case is called and just after stopping the sampling thread, the *finishCase()* method of test case is called. Just prior to starting the collection of metrics for a given run, *setupRun(object[] runParams)* is called, and just after collection has completed for a run the *finishRun()* method of the test case is called. These methods facilitate setup and finalization of each run as well as the set of runs for a given test case.

4.3 Framework Implementation

The framework is implemented as a collection of extendable and reusable classes. Figure 4.2 shows a class diagram for the DTPT Framework. This figure is organized to mirror the collaboration diagram structure so that it is easy to see how the classes below are involved in the process shown above.

The framework consists of the following major components:

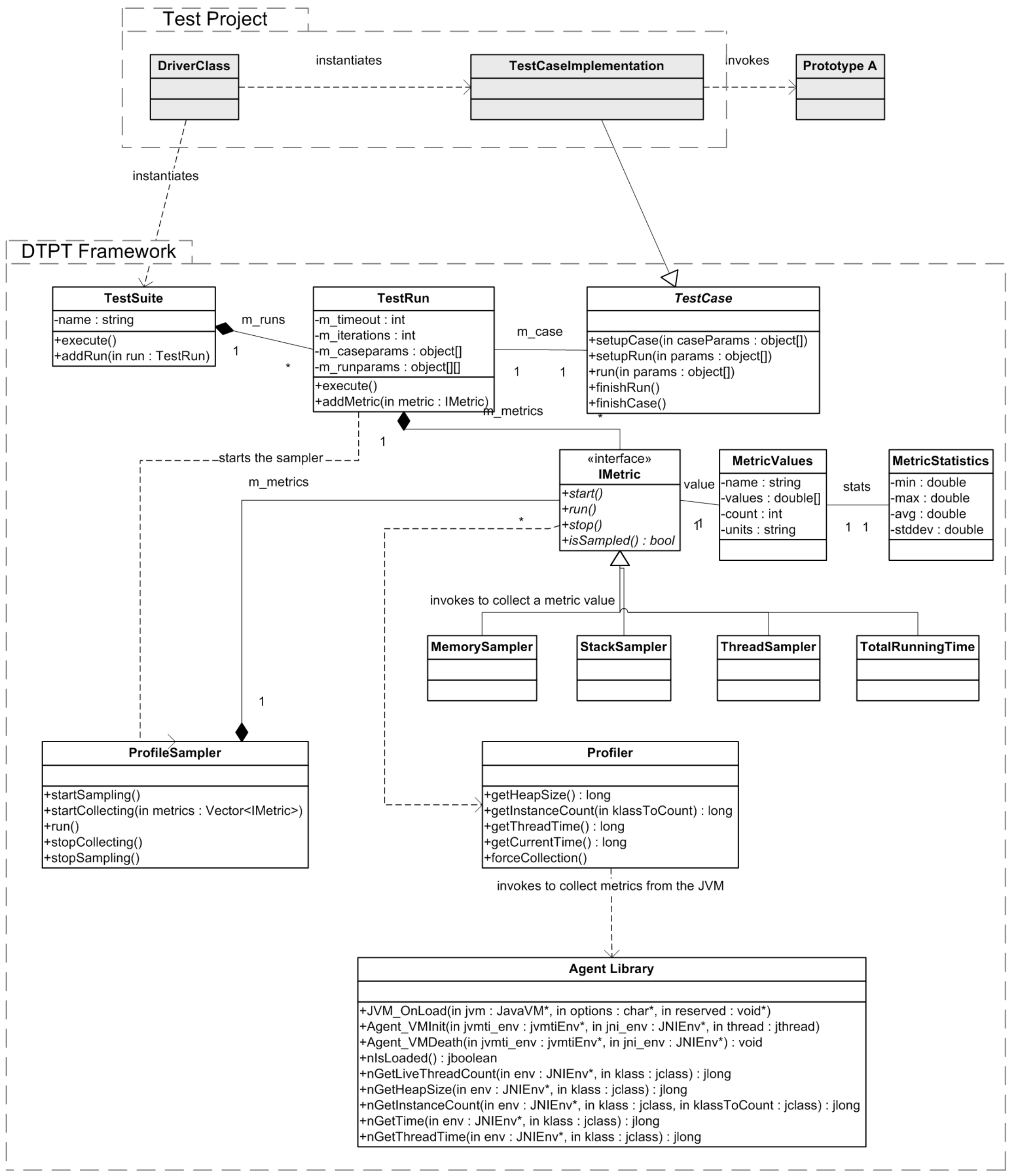

Test Project
DriverClass
instantiates
TestCaseImplementation
invokes
Prototype A
instantiates
DTPT Framework
TestSuite
-name : string
+execute()
+addRun(in run : TestRun)
m_runs
1
*
TestRun
-m_timeout : int
-m_iterations : int
-m_caseparams : object[]
-m_runparams : object[][]
+execute()
+addMetric(in metric : IMetric)
m_case
1
1
TestCase
+setupCase(in caseParams : object[])
+setupRun(in params : object[])
+run(in params : object[])
+finishRun()
+finishCase()
m_metrics
*
1
starts the sampler
m_metrics
«interface»
IMetric
+start()
+run()
+stop()
+isSampled() : bool
*
value
1
MetricValues
-name : string
-values : double[]
-count : int
-units : string
stats
1
1
MetricStatistics
-min : double
-max : double
-avg : double
-stddev : double
invokes to collect a metric value
MemorySampler
StackSampler
ThreadSampler
TotalRunningTime
1
ProfileSampler
+startSampling()
+startCollecting(in metrics : Vector<IMetric>)
+run()
+stopCollecting()
+stopSampling()
Profiler
+getHeapSize() : long
+getInstanceCount(in klassToCount) : long
+getThreadTime() : long
+getCurrentTime() : long
+forceCollection()
invokes to collect metrics from the JVM
Agent Library
+JVM_OnLoad(in jvm : JavaVM*, in options : char*, in reserved : void*)
+Agent_VMInit(in jvmti_env : jvmtiEnv*, in jni_env : JNIEnv*, in thread : jthread)
+Agent_VMDeath(in jvmti_env : jvmtiEnv*, in jni_env : JNIEnv*) : void
+nIsLoaded() : jboolean
+nGetLiveThreadCount(in env : JNIEnv*, in klass : jclass) : jlong
+nGetHeapSize(in env : JNIEnv*, in klass : jclass) : jlong
+nGetInstanceCount(in env : JNIEnv*, in klass : jclass, in klassToCount : jclass) : jlong
+nGetTime(in env : JNIEnv*, in klass : jclass) : jlong
+nGetThreadTime(in env : JNIEnv*, in klass : jclass) : jlong

Figure 4.2: Class structure of the Java Framework and C++ Agent library.

1. Designers interact with the DTPT Framework through a **Java library**. The library contains a number of packages for building and executing test cases, collecting performance metrics, and formatting results. Because the framework is lightweight and separated into a number of simple Java classes, designers can extend the framework as needed.

2. The Java Virtual Machine Tools Interface (JVMTI^{TM}) is an interface to the virtual machine that allows native libraries to instrument the virtual machine [46]. This method of instrumentation replaces the previous Java Virtual Machine Profiler Interface (JVMPI), and introduces much less overhead that its predecessor. The Framework uses a **JVMTI^{TM} Agent** to collect some performance metrics directly from the virtual machine. This portion of the framework is written in C++, but since the majority of its functionality is interacting with the (JVMTI^{TM}), it can be compiled on multiple platforms. The C++ code was originally developed and tested on a Apple PowerMac G5 under Mac OS X version 10.4 and has also been compiled and tested on a Intel Core 2 Due machine under Windows XP with MinGW [32].

3. Design test suites in the framework can be invoked using the **command line**. Since use of the framework can be controlled from the command line, execution of design test suites can be easily automated or integrated with existing tools.

4. **Test suites** represent design scenarios and are composed of Test Runs. A test run corresponds to a set of run parameters and a given test case. Test cases can accept a number of parameters, one of which is generally the prototype to execute the test case against. Test cases describe the scenario to test in a number of phases: setupCase, setupRun, run, finishRun, finishCase. The details of these phases are explored further in the rest of this chapter.

5. Design test suites specify the **performance metrics** that should be collected at runtime when executing each test case. While the framework includes some common performance metrics, users can build custom performance metrics as well.

6. Metric results are captured during test execution and then can be fed through a **result formatter** to produce documentation (tables and graphs) to facilitate easy comparison between alternatives.

7. While test cases utilize facilities of the framework, the **prototypes under test** do not require modification to be tested within the framework. In this way, prototypes can use the facilities of existing software (like components, web services, or middle-ware) to reduce prototype development time.

4.4 Repeatable results

While the process suggests that repeating an experiment should give the same result, in the implementation this is unlikely. While the framework implementation uses a function for this mapping, one of the inputs – the environment – is difficult to control in an actual system. Since the implementation is running on a multi-tasking, pre-emptive operating system on a system with multiple execution cores, the system state is complex. Because of the environment's complexity, I generally do not expect test case results to be exactly the same over separate executions as the environment will be at best similar. Rather, in the implementation, I hope that multiple executions of the same test give consistent results such that they can inform the decision to be made, even if they vary slightly.

4.5 Functionality

4.5.1 Sampled metric collection

Just before the framework executes a test case it activates the metric sampling thread. This thread runs periodically (the period can be adjusted manually) and collects each metric enabled for the test run.

There are two main approaches to collecting performance metrics: collecting periodic samples as our framework does; and instrumenting the system under test with code to collect performance metrics as DTrace and many other profilers do. In this framework, I have chosen sampling since in general it introduces less overhead and can be enabled and disabled dynamically while the system under test is running. While DTrace uses instrumentation and avoids both of these problems, it requires users to learn a domain-specific language called D to specify active profile sites, and uses a system of replacing machine language instructions in-memory with profile trampolines to collect performance metrics [10]. Thus, while sampling has disadvantages, explored further in section 4.7, I have chosen sampling since it is easy to implement and – more importantly – easy for test-writers to understand and use.

4.5.2 Test iterations

Users can indicate for a given test case how many iterations to execute. By gathering metrics over multiple iterations, one can simulate scenarios in which a given situation is executed multiple times and in so doing capture the performance effects of, for example, dynamic optimization. Furthermore, one can specify whether to include the first iteration in the total timing to help developers avoid start-up effects. These features are adjustable from the test suite setup since start-up behavior may be important in some design scenarios and not in others.

4.5.3 Parameterized tests

Test cases accept parameters to their *setupCase*(), *setupRun*(), and *run*() methods. These parameters give test cases the flexibility to exercise a number of scenarios by simply changing these values. For example, a test case might accept an algorithm as a parameter if we are interested in testing alternate algorithms to solving a given problem.

To help automate the execution of parameterized tests, a test run is initialized with a matrix of test case parameters. When executed, the test run will execute the test case for each combination of the run parameters contained in the matrix.

4.5.4 Metric statistics

Once a test case has executed, the framework calculates statistics over the metric values collected including: minimum, maximum, average, and standard deviation. While these statistics may not be useful for all metrics, it is convenient to have them available.

4.5.5 Results documentation

After a test suite has executed a given test run, the metric results can be passed to a result formatter. Result formatters produce human readable representations of the collected data. I have implemented two result formatters: one to produce tabular representations of collected metrics (exported as HTML table fragments), and one to produce graphs of metric values over time (exported using gnuPlot as portable network graphics (PNG) files).

4.5.6 Time-limited test cases

User can specify a maximum execution time for each test case, after which they are terminated and a timeout is recorded. This is valuable for comparing implementations whose performance may degrade significantly for certain inputs. If we were to compare sorting algorithms for a variety of input sizes, we may wish to put a maximum execution time limit on the test case to cut off slow sorts (like bubble sort and insertion sort) for large input sizes while still providing a meaningful comparison of sorts for input sizes where their execution time is within this maximum time limit. Figure 4.3 shows this example with a maximum execution time of five seconds; for those input sizes where the test cases timeout, no performance metrics are reported on the graph.

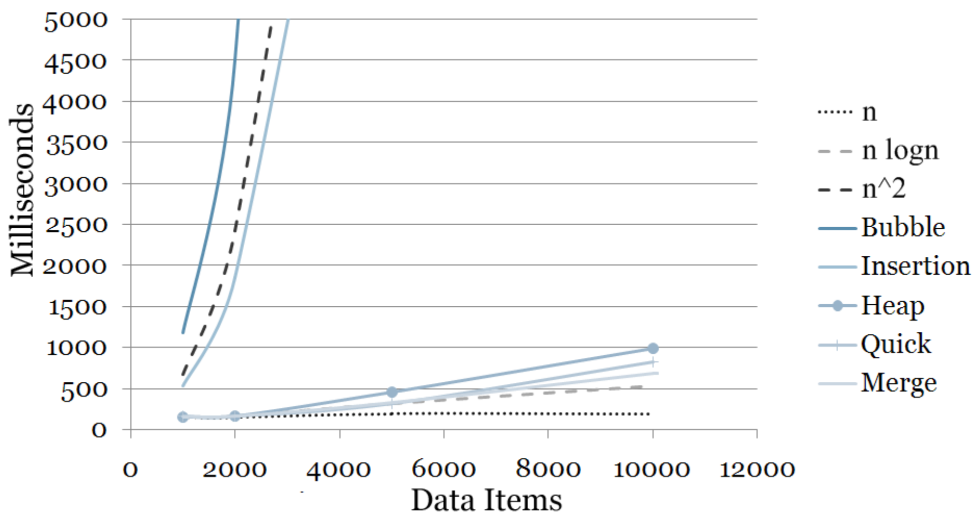


Figure 4.3: An example comparison of sorting algorithms using a maximum execution time to cut off slow sorting algorithms while providing a meaningful comparison of each algorithm.

4.6 Representing test cases

A test case is represented as a class that inherits from the abstract base class *TestCase*. While I could have used extensible markup language (XML) or a domain specific language (DSL) to capture test cases, I have followed the industry-standard convention introduced by JUnit [34]. This also allows test case writers to rely on their knowledge of Java programming and their programming tools to build test cases (e.g. using features like method auto-complete, and compile-time syntax checking).

4.6.1 Specifying metrics

The framework supports both *sampled metrics*, those which are collected periodically throughout execution, and *marked metrics*, those which are collected only at the beginning and end of execution periods. The framework provides an interface *IMetric* which all metrics must inherit from as well as four concrete metric classes described below. The source code for *IMetric* is shown in Figure 4.4.

- **MemorySampler** samples the size of the Java Heap, measured in kilobytes, periodically. This metric can be used to compare heap memory usage.
- **StackSampler** captures the stack depth, measured in stack frames, periodically. One can look at how recursive algorithms use the stack over time.
- **ThreadSampler** samples the number of actively running threads. This metric can help

designers compare multi-threaded implementations.

- **TotalRunningTime** is a marked metric and simply captures the total running time, measured in seconds, for a given execution period. One can use this metric to compare the total execution time of alternate implementations.

```
public interface IMetric {
        public boolean isSampled();
        public void start();
        public void run();
        public void stop();
        public String name();
        public MetricValue value();
}
```

Figure 4.4: The IMetric interface; part of the DTPT Framework.

4.7 Implementation criticisms

One clear fault of this implementation is that performance metrics are collected from a thread that is running simultaneously with the test being executed. This sampling thread introduces overhead and may interfere with the natural behavior of an application (the thread switch will invalidate some of the cache, and use processing cycles). Thus, by sampling the test while it executes, the framework changes the performance behavior of that test case. While reducing the rate at which the application is sampled will reduce the overhead introduced, it will also reduce the accuracy of metrics collected.

The second criticism of the framework implementation is that the accuracy of metrics is partly dependent on the sampling frequency, and thus some error is introduced by sampling periodically rather than using continuous monitoring (via hardware performance counters). Conversely, this inaccuracy can be reduced by increasing the sample rate.

Since the level of metric accuracy and introduced overhead may depend on the situation being tested, the framework implementation allows users to configure the sampling frequency. While this does not alleviate both criticisms it allows users to mitigate one at the expense of the other. Furthermore, the sampling thread of the framework is rather light-weight, only waking to collect the enabled performance metrics then returning to sleep.

4.8 Further implementation ideas

In the future, I would like to extend the metrics included with the framework. While developers can create new metrics, I would like to include a few rather interesting metrics like: data transfer (over a network interface or from a disk), degree of parallelism (on a multi-core system), garbage collection overhead, and power consumed (for mobile applications).

While this implementation of the framework only includes sampled metrics, I would like to introduce metrics that could be collected by weaving aspects into the implementation at load time. This would allow designers precise control over where instrumentation is added, reducing the overhead of collecting metrics and increasing the accuracy with which metrics are collected.

4.9 Using the framework

To use the framework, a designer would go through the following process:

1. **Create a new project** that will contain the test suite and test cases.

2. **Add classpath references** to the prototype implementations to be tested as well as the framework library.

3. **Build one or more test cases** that inherit the abstract class *TestCase* in the framework library. The test case(s) must implement the *run(Object[] params)* method, but can also override *setupCase(Object[] caseParams)*, *setupRun(Object[] params)*, *finishRun()*, and *finishCase()*.

4. **Build a driver class** with a *static void main(String[] args)* method. This class should instantiate a *TestSuite* instance, and add *TestRun* instances to the test suite.

5. **Setup each test run** with the test cases implemented in step 3. The performance metrics, case parameters, and run parameters matrix should be added to each *TestRun* instance.

6. **Pass the results to a result formatter**. The result formatter will produce human readable documentation of the test results.

7. **Compile and execute the driver class**. When running the driver class, the Java Virtual Machine needs to know about the framework agent library. This can be accomplished with either the *agentlib* or *agentpath* command line options to the JVM.[1]

Chapter 5 presents a set of experiments designed to verify that software designers using the framework and process can produce results that are consistent with our intuition and standard performance rules. Chapter 6 then presents a set of experiments that suggest the DTPT process and framework are valid approaches to the problem of design-time performance tesing.

[1]For more information on using agent library command line options in Java, see: http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html#starting.

Chapter 5

Verification Experiments

A framework for performance comparison should provide accurate results so users will trust its output. In this chapter I start by introducing an example test suite, test case, and prototype to show how the framework is used. I then explore, through a set of experiments, the accuracy of the framework by comparing the results obtained to expert advice and our intuition of well known best performance practices. Next I establish that the overhead introduced by using the framework for a set of test cases is reasonable. With these experiments I intend to illustrate that the results given by the framework are consistent with previous work, match our expectations, and include a only reasonable overhead.

5.1 Sorting

Sorting a sequence is a fundamental operation in computer science that is explored in any introductory algorithms textbook (for example, see Corman et al.'s Introduction to Algorithms [13]). There are a number of algorithms one can use to sort a sequence each with a well known timing complexity; some of the most common sorting algorithms and their average case timing analysis are shown in Table 5.1 and Figure 4.3. Because sorting is a familiar problem and the performance of many approaches is known, I have used it here to introduce how test cases can shed light on performance.

The process for design-time performance testing in this thesis is intended to compare design decisions, yet choosing a sorting algorithm is a trivial design decisions since an optimized quick sort is sufficient for most applications and changing the sorting algorithm is transparent to the rest of the application. Nevertheless, sorting does illustrate the use of the framework, and the process can be used to expose the performance trade-offs of various algorithms. The later examples in this chapter present more substantial design decisions that designers would get a real benefit from prototyping using the design-time performance testing process.

To compare the algorithms in Table 5.1, we must build a test suite to represent the scenarios we wish to test, build a test case to execute each prototype, implement the prototypes to compare, execute the test suite, and analyze the performance results. Each step is illustrated for the sorting

Table 5.1: Average case algorithmic complexity of algorithms for sorting a sequence of n items.

Algorithm	Average Case Timing Bound
Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Heap Sort	$O(n \log n)$
Merge Sort	$O(n \log n)$
Quick Sort	$O(n \log n)$
Radix Sort	$O(n \log_2 k)$ [a]

[a]Where k is the largest integer in the sequence to be sorted.

example in the following subsections:

5.1.1 Test Suite

To begin, we create a Driver Class that can be invoked from the command line, and instantiates and executes a *TestSuite* object. The test suite contains a collection of *TestRun* objects. A test run specifies what metrics to collect (in this case, just the TotalRunningTime metric) as well as the value for case and run parameters. A *TestRun* accepts a jagged matrix of run parameters; this allows users to automatically execute each combination of run parameters without having to specify each combination explicitly. The following pseudocode shows how the Sorting Driver sets up the metrics, test cases, and prototype implementations and executes the test suite. Listing A.1 shows the full Java source code for this example.

Algorithm 1 Sorting test suite

```
1: $s \leftarrow$ new $TestSuite$
2: for $r \in TestRuns$ do
3:     for $m \in PerformanceMetrics$ do
4:         $r.addMetric(m)$ {add each of the metrics to the test run}
5:     end for
6:     $s.addTestRun(r)$ {add each of the test runs to the test suite}
7: end for
8: for $p \in Prototypes$ do
9:     $s.addPrototype(p)$ {add each of the prototype implementations}
10: end for
11: $s.execute()$ {execute the test suite}
```

5.1.2 Test Case

A test case inherits the framework base class *TestCase* and must override the *run(Object[] params)* method. The *run* method of a test case executes some scenario in the prototype. During execution of the *run* method, the framework will be collecting performance metrics. In the sorting example, the run method generates a sequence of random integers, and sorts that sequence using the prototype. Pseudocode for this example sorting test suite is below, and a full Java implementation in the framework is available in Listing A.2.

Algorithm 2 Pseudocode for the sorting test case

```
1:  for $p \in Prototypes$ do
2:      $l \leftarrow []$
3:      for $i \leftarrow 1...count$ do
4:          $l[i] \leftarrow []$
5:          for $j \leftarrow 1...length$ do
6:              $l[i][j] \leftarrow random(1, max)$
7:          end for
8:          $p.sort(l[i])$
9:      end for
10: end for
```

The Java source code for this example test employs two conventions I have found useful when building framework tests. First, the *run* method simply typecasts parameters and passes them to a private, typed method that actually executes the test case. This way type casting errors are found early and at a single location within the test case. Second, the prototype to test is passed as one of the run parameters and all prototypes implement a common interface. In this way, the test case is generic and can execute any of the prototypes in the same way. It also allows us to vary the prototype easily by passing in each prototype to the run parameters matrix.

5.1.3 Prototype

The prototype implements a common interface, *Sorter* in this example, so that any prototype can be exercised with the same test case. Within the prototype, we execute the requested operation in a way that approximates a full implementation. In the sorting example, each prototype implements the sorting algorithm, but as we will see in later examples, our prototypes need not be complete implementations, they only need to approximate the run time performance of the full implementation.

The quicksort implementation is available in Listing A.3. The example prototype implements the Quick Sort algorithm. While this implementation is complete, it selects a random pivot and as such may select poor pivots; this is acceptable even if the full implementation might use a more advanced pivot selection algorithm since the prototype need only be an approximation. Furthermore, we can refine the prototype later to include a more advanced pivot selection algorithm or create another set of prototypes to compare pivot selection algorithms.

5.1.4 Command line usage

Figure 5.1 illustrates how to invoke the Java Virtual Machine, reference the framework agent library (named *dptp*), include the prototype and framework classes in the classpath, and start executing the *public static void main(String args[])* method of the Driver class. The agent library *cbprof* (libdptp.jnilib or dptp.dll) needs to be installed to the default agent library directory.

```
java -agentlib:dptp -classpath [1];[2] SortingDriver
```

Figure 5.1: To invoke the SortingDriver, the user will need to replace [1] and [2] with the paths of the framework and prototype class files respectively. Alternatively, if using the Eclipse Integrated Development Environment (IDE), one can reference these projects from the testing project.

Executing this command will produce the results shown in Figure 4.3.

5.2 Experiments

5.2.1 Environment

All experiments were conducted on an Intel® Core^{TM} 2 Duo T7200 2.0 Ghz computer with 1 GB of RAM under Windows XP Pro SP2. All experiments were developed and executed in the Eclipse Europa IDE on the virtual machine provided by Sun Microsystem's J2SE jdk1.6.0_05.

5.2.2 Procedure

The first step in executing these experiments was to find a number of intuitive performance best practices. I drew best practices from three sources:

1. John O'Hanley's website of Java best practices [33]; I focused on the subset of those motivated by performance consequences.

2. The Eclipse Foundation's list of performance bloopers encountered by Java plug-in writers [36]; I focused on the subset of bloopers not specific to Eclipse plug-in development.

3. Introduction to Algorithms textbook by Corman et al. [13]; I compared seven sorting algorithms.

The second step was to prototype those best practices along with their naive alternatives (or each of the algorithms in the case of sorting). To facilitate comparison, I wrote design test suites for each best practice to compare the total running time and heap memory usage of each prototype. I confirmed the correctness of each prototype and ensured that each alternative prototype produced the same output for a number of input cases using JUnit prior to running the design test suites.

The prototypes for this experiment were small Java classes: the bloopers prototypes are 484 lines, while the sorting routines are 660 lines.

5.2.3 Results and Analysis

Table 5.2 illustrates the experiment results found for those best practices identified by O'Hanley and the Eclipse Foundation. Figures 5.2 and 5.3 show the total timing and memory usage of each sorting algorithm.

Best Practices and Bloopers

Boxing Primitives involves a programmer either explicitly or implicitly wrapping a primitive data type (*byte, short, int, long, float, double, boolean, char* in Java) using a corresponding reference type. For example, when passing an integer to a method expecting an *Object* (since all reference types derive from *Object*), the programmer must explicitly wrap the *int* using an *Integer* instance, or allow the compiler to implicitly do this boxing. The results were obtained by varying the total number of method calls from 50,000 to 1,000,000.

The test case in this situation compares two methods for adding integers, one which boxes two operands and the return value, and the other which accepts and returns primitives. The results show, as O'Hanley predicted, that boxing incurs time and heap overhead since each method call involves instantiating three temporary heap objects that must be garbage collected later.

Exception Throwing involves leaving ordinary control flow to signal an extraordinary situation (e.g. invalid argument values, arithmetic overflow). The results were obtained by varying the probability of 1,000 method calls encountering an error from 0.005 to 0.5.

This test case compares throwing an exception to simply returning an integer error code. One can see that throwing exceptions not only introduces heap overhead (storage of the *Exception* and *StackTrace* objects), but also significantly increases total running time.

String Concatenation involves creating a new string by combining two or more strings. While concatenation can be accomplished with the + operator, Java offers the *StringBuffer* class to programmers who need to concatenate large strings. Java *String* objects are immutable while *StringBuffer* objects are mutable. The results shown were obtained by building a one-page document by concatenating string chunks. The size of each chunk was varied from 1 character to 1024.

The StringBuffer reduces the amount of heap memory used when concatenating short strings since each concatenation to a StringBuffer does not incur creating a new String object as in the naive approach. When concatenating strings in small chunks, the StringBuffer clearly improves performance as O'Hanley and the Eclipse foundation predicted, but when concatenating large chunks, the total time required is nearly identical for either approach.

Substring Extraction involves copying a small substring from a much larger string. To support fast substring extraction, calling the *substring* method of the *String* class creates a new *String* instance that references a portion of the original larger string's underlying buffer. If we are extracting small strings from a large string, keeping the large string around may introduce significant heap overhead. To prevent this behavior, we can explicitly call the string constructor on the substring method's return value to copy the substring into a new underlying string buffer, allowing the larger string's buffer to be garbage collected.

The results were obtained by extracting ten substrings of large strings (10,000 to 100,000 characters). The size of these substrings varied from 10% to 50% of the overall string length. The results show that while the default substring behaviour may be faster, we can save significant memory by explicitly copying the underlying buffer as predicted by the Eclipse Foundation.

Buffered IO reads data from the underlying IO device in large chunks to reduce IO operations, but buffers that data so the programmer can read data in any size chunk they wish. I conducted the experiment for two situations using different IO devices: downloading a file from the Internet (the network card), and reading a file (hard drive). For both situations, the chunk size read varied from a single byte to 1024 bytes.

For File IO, the results clearly indicate that buffering significantly reduces the total time required to read the file when reading in small chunks; consistent with O'Hanley and the Eclipse Foundation. Network IO does not exhibit this same, expected result. This test uses TCP/IP to read data, a protocol which includes buffering, and is facilitated through a series of drivers which make up the network stack. Due to TCP/IP buffering, and details of the network stack, buffering does not provide any consistent benefit when reading network data.

Furthermore, one will notice that although buffering introduces some temporary heap storage, the unbuffered test uses more heap spaces. There are two possible explanations: first, many devices do not support reading single bytes, so single byte reading is mimicked using internal buffers (which take heap space), and since each method call may block, far more synchronization objects will be instantiated when reading small chunks without a buffer.

Sorting

Sorting is a common, well-known problem in computer science. There are a variety of approaches to sorting an array of values, each with well-understood timing and memory bounds. To validate the results given by the framework, I wanted to compare the total running time and heap memory consumption for a number of sorting algorithms. I implemented six sorting algorithms: Bubble, Insertion, Selection, Quick, Heap, Merge, and Bucket. I wrote unit tests to ensure the correctness of these algorithms, then wrote a design test suite to compare their respective performance and the performance of the built-in, static *sort* method of the *java.util.Arrays* class.

Figure 5.2 graphs the total time of each sorting algorithm. This graph clearly illustrates the difference between the simple $O(n^2)$ algorithms like Bubble, Insertion, and Selection and shows that all of the $O(n \log n)$ algorithms run in similar time, while Bucket sort runs slightly faster $O(n(N/b))$ where N is the maximum value and b is the number of buckets. The built-in sorting algorithm is an optimized quicksort [1] and runs the fastest for all input sizes. This optimized quicksort includes the following optimizations not present in my implementation: degrading to insertion sort for small sublists (under 10 items), and selecting the best pivot point from a set of nine potential pivots.

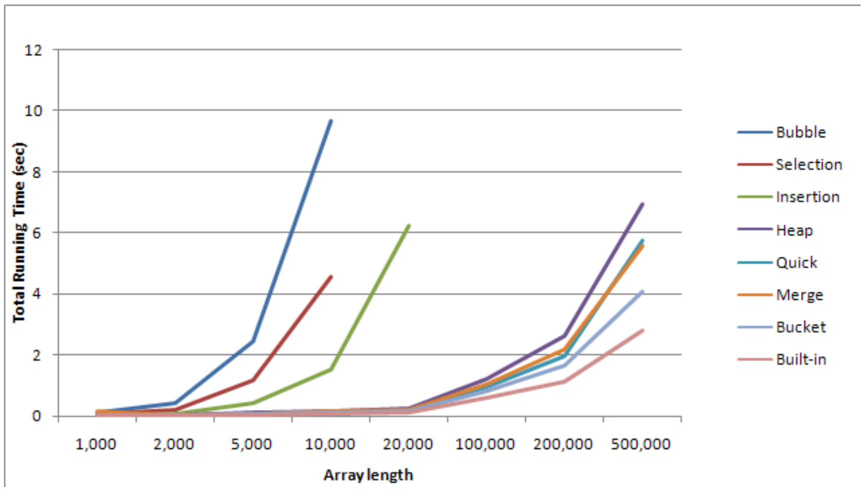


Figure 5.2: Total running time of each sorting algorithm for various input array lengths.

The heap memory usage for each sorting algorithm graphed in Figure 5.3 clearly shows the difference between sorting in-place and using $O(n)$ additional storage. While Bucket and Merge both require an additional array to store sorted values, Bucket also requires an array to store the size of each bucket. The heap memory usage of Heap and Quick overlap almost exactly as they both sort their input in-place.

[1]Source code for *java.utils.Arrays.sort(int[] a)* method in the GNU class path http://www.docjar.com/html/api/java/util/Arrays.java.html

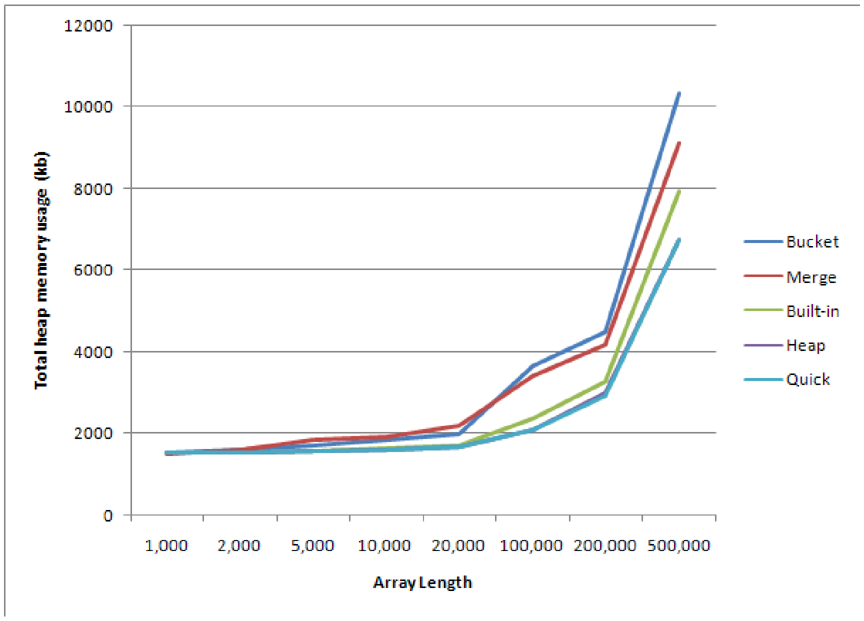


Figure 5.3: Total heap memory usage for five fastest sorting algorithms.

5.3 Overhead Experiments

Instrumenting a running application introduces overhead. The purpose of this experiment is to understand the overhead introduced by the DTPT Framework when measuring performance results. Because we are trying to estimate performance by instrumenting prototypes, minimizing the overhead introduced by the framework is important so we do not obscure the performance results.

The amount of overhead introduced from instrumentation can vary depending on the application being instrumented; thus, we can not predict the performance overhead users will experience in all situations. To ensure this experiment captures the overhead of realistic situations, I have run it over each of the best practices test cases.

5.3.1 Procedure

Examining the overhead of performance measurement is not an easy task; we are attempting to measure the cost of measurement. In this experiment, I measure overhead with respect to two quantities: total running time, and heap memory usage. To compare the overhead, I ran a design test suite consisting of the best practices introduced in the previous section once with the framework active, and once with the framework disabled.

Total running time was calculated by comparing the difference between calls to the built-in *System.currentTimeMillis()* before and after the execution of each test case. While this method is not as accurate as the JVMTI method *GetTime*, calling this method would require loading the agent library (one of the costs we are attempting to measure).

The framework introduces heap memory overhead in two ways: first, the overhead introduced by starting a sampling thread, and second, the overhead of storing metric results on the heap. Without starting a sampling thread, we cannot measure the usage of heap memory as the application executes. To estimate the heap memory overhead of the framework, we measure the size of metric storage on the heap when the framework is active. This is underestimates the total overhead (since it does not include the overhead of the sampling thread), but this measurement introduces very little overhead of its own.

Figure 5.4 shows an example of the output produced by the framework. Note that the framework estimates and outputs the overhead required to store metric values.

```
 Running case:  edu.usask.cs.hopkins.tests.bloopers.NetworkIO,
www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF, 1024, true
TotalRunningTime:  4.391 sec
MemorySampler:  527.24 avg kb
Overhead:  14 kb
Running case:  edu.usask.cs.hopkins.tests.bloopers.NetworkIO,
www.bredemeyer.com/pdf_files/ArchitectureDefinition.PDF, 1024, false
TotalRunningTime:  4.266 sec
MemorySampler:  519.42 avg kb
Overhead:  14 kb
```

Figure 5.4: Shows an example of the framework output for two test runs.

While we could have used an external utility to monitor memory usage and total running time such as Linux's *top* command, this type of utility introduces overhead of its own.

5.3.2 Results and Analysis

The total running time overhead is rather low in all situations; Boxing Primitives has a high percentage overhead, but this is because each experiment takes so little time to execute (less than 0.5 seconds on average). Calls to the agent library are simple both in terms of data transfered and time complexity. *nGetTime* and *nHeapSize* both take no parameters and return only a *long*, and *nGetTime* involves only a single call to the JVMTI interface. *nHeapSize* iterates over the objects in the heap, but since our experiments generally use only a small part of the heap, this call should be fast as well.

The total heap memory overhead is also very low. Metric values are stored in an array of doubles, which grows as needed, rather than an array list which would involve boxing and significantly increase the total heap memory overhead even though the underlying storage mechanism is an array. The memory overhead found in Buffered Network IO and File IO are higher because these experiments run longer and thus have more measurements. This can be reduce by changing the sampling frequency for long test runs.

This experiment does not capture all the heap memory overhead involved in running the framework. The framework spawns a sampling thread, which introduces some objects on the heap (e.g. an instance of *Thread*). Loading the profiler library also introduces some overhead since this shared library must be loaded into memory and store some local data.

The framework can run in real time, as opposed to running as a simulation like Javana [31], which is important for quickly building and comparing prototypes of alternatives. Memory overhead is estimated and reported by the framework and this overhead can be adjusted by changing the sampling frequency.

5.4 Conclusion

In this chapter, I described the usage of the framework through a sorting test case example including pseudocode and full Java implementations of a test suite, a test case, and a prototype implementation. I also showed the results of various experiments to verify the accuracy of the framework when compared to expert advice and our intuition. Furthermore, I showed that the overhead introduced by using the framework is reasonable for a number of test cases. In the next chapter I look at how the framework can be used in a real application and whether the results produced by testing prototypes are good estimates of full production implementations.

Table 5.2: This table compares the time and memory trade offs between naive implementations and best practices.

	Time (sec)		Memory (kb)	
	min	max	min	max
Boxing Primitives				
Primitive Data	0.047	0.75	215	221
Boxed Data	0.093	1.125	436	921
Improvement	49.5%	33.3%	50.7%	76.0%
Exception Throwing				
Error Code	0.281	0.312	305	331
Try / Catch	0.328	2.437	678	917
Improvement	14.3%	87.2%	55.0%	63.9%
String Concatenation				
StringBuffer	0.000	0.031	303	302
String	0.000	0.312	294	754
Improvement	0.0%	90.1%	-3.1%	59.9%
Substring Extraction				
New String	0.000	0.141	287	809
Reference	0.000	0.156	304	1,436
Improvement	0.0%	9.6%	5.6%	43.7%
Buffered Network IO				
Buffered	20.875	22.81	702.66	978.60
Unbuffered	25.734	23.765	729.87	799.70
Improvement	18.9%	4.0%	3.7%	-22.4%
Buffered File IO				
Buffered	0.016	1.157	215	247
Unbuffered	0.031	21.157	215	416
Improvement	48.4%	94.5%	0.0%	40.6%

Table 5.3: Illustrates the overhead introduced by the framework in terms of total running time and heap memory usage for each of the performance bloopers tests executed in the previous section.

	Time		Memory	
	sec	%	kb	%
Boxing Primitives				
Overhead	0.02	17.14	0.00	0.00
Exception Throwing				
Overhead	0.00	3.80	0.05	0.06
String Concatenation				
Overhead	0.00	3.80	0.00	0.00
Substring Extraction				
Overhead	0.00	0.00	0.00	0.00
Buffered Network IO				
Overhead	1.32	5.14	25.72	3.83
Buffered File IO				
Overhead	0.06	2.65	4.00	1.07

Chapter 6
Validation Experiments

In this chapter I argue that the proposed process and framework are valid approaches for design-time performance testing. I show that we can use prototypes to understand the performance consequences of real design decisions. The design decisions in these experiments are drawn from the I-Help Discussion System so I begin with an explanation of that system and an explanation of the prototype used in the experiments. I then explain the design decisions explored in each experiment, then present the experiments. The experiments consist of running the test cases on a prototype implementation and a full scale implementation to show that the performance estimates produced by simple prototypes lead to decisions consistent with the performance of a full-scale implementation.

6.1 I-Help Discussion System

6.1.1 System overview

The I-Help Discussion System is a web-based application developed by the ARIES Laboratory in the Department of Computer Science at the University of Saskatchewan [24]. The primary function of this application is to facilitate on-line discussion between students and staff within a given course. The Department of Computer Science has developed and supported the I-Help system for over six years. In the Spring 2008 semester, I-Help has been used by thousands of users in about 20 courses, with each course having 50 - 1,000 discussion posts.

The I-Help Discussion System is in active use and as such it provides an interesting test bed for the process and framework of my thesis. I have selected the I-Help system for the following reasons:

- The student, staff, and faculty who use I-Help have an **expectation of system up-time**. While not a mission critical system, updating the system to try a number of different options could adversely affect the system users and any down-time (due to a faulty update) would be best avoided.

- The **architecture of I-Help is non-trivial**. I-Help uses an n-tier architecture with a

Relation Database Management System (RDBMS) for storing persistent data, Java classes to implement the business logic layer, Java Server Pages (JSPs) to implement the presentation layer, and a Java application server to distribute web requests among Java objects. I-Help also uses Asynchronous Javascript and XML (AJAX) to provide an interactive web-based interface.

- Since I-Help is used by a large number of students and courses, it has a **significant workload** it must service. Thus, performance is an important concern for the designers of the I-Help Discussion System.
- The I-Help system is written in **Java** and thus can be instrumented using the Java framework I have developed.

Based on an analysis of the I-Help Discussion System's requirements, purpose, and implementation I have come up with three significant design changes that will likely have significant performance impacts. To test these changes, I have developed a prototype system that approximates the I-Help Discussion system for the purpose of comparing performance trade-offs. The next section describes this prototype, then I discuss each design change and introduce example test cases that are used to approximate the performance consequences of each change.

6.1.2 Prototype

I developed a prototype that implements the core functionality of the I-Help discussion system. This prototype is simple and approximates much of the system functionality, but the hope is that I can quickly compare design alternatives in this prototype (since it is not very complex, my changes should be simple), and collect performance metrics for these alternatives informing design decisions before going through the process of building a full implementation.

In the case of the I-Help discussion system, the implementation already exists and I could have used this implementation as part of the prototypes. By building a simple prototype I illustrate how a designer could apply the framework to a system for which no implementation exists. Using simple prototypes also means the design changes are easier to make, but may be less accurate than adapting the existing implementation. Finally, by using simple prototypes, the potential design changes are not restricted by the limitations of the current implementation.

In building a prototype, rather than relying on the existing implementation, we are also forced to examine the assumptions present in the prototype. The prototype I-Help discussion system and related test cases have a number of simplifications not present in the actual I-Help system:

- The popularity of discussion posts follows a Zipf-like distribution. Breslau et al. found that this was true for web documents in general, so it is likely a good approximation [8].

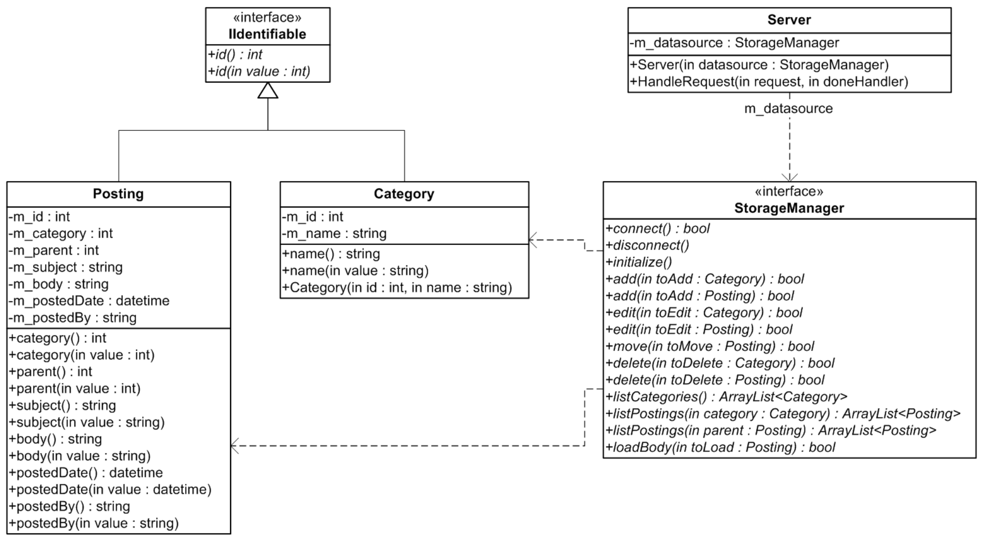

«interface»
IIdentifiable
+id() : int
+id(in value : int)
Server
-m_datasource : StorageManager
+Server(in datasource : StorageManager)
+HandleRequest(in request, in doneHandler)
m_datasource
Posting
-m_id : int
-m_category : int
-m_parent : int
-m_subject : string
-m_body : string
-m_postedDate : datetime
-m_postedBy : string
+category() : int
+category(in value : int)
+parent() : int
+parent(in value : int)
+subject() : string
+subject(in value : string)
+body() : string
+body(in value : string)
+postedDate() : datetime
+postedDate(in value : datetime)
+postedBy() : string
+postedBy(in value : string)
Category
-m_id : int
-m_name : string
+name() : string
+name(in value : string)
+Category(in id : int, in name : string)
«interface»
StorageManager
+connect() : bool
+disconnect()
+initialize()
+add(in toAdd : Category) : bool
+add(in toAdd : Posting) : bool
+edit(in toEdit : Category) : bool
+edit(in toEdit : Posting) : bool
+move(in toMove : Posting) : bool
+delete(in toDelete : Category) : bool
+delete(in toDelete : Posting) : bool
+listCategories() : ArrayList<Category>
+listPostings(in category : Category) : ArrayList<Posting>
+listPostings(in parent : Posting) : ArrayList<Posting>
+loadBody(in toLoad : Posting) : bool

Figure 6.1: The classes in the I-Help prototype implementation. The implementation is simplistic, which allows us to quickly prototype and test alternatives.

- The test cases use an actual discussion tree scraped from a popular discussion website.
- The db test suite is multi-threaded to approximate the behavior of a production web server. Since requests will be handled in parallel, the underlying data sources need to handle this correctly.

While my prototype implementation includes the assumptions above, one could easily modify the system to use a different set of assumptions (e.g. by incorporating discussion and request data gathered from the production system). In fact, for deciding between promising alternatives with similar initial results, designers should refine the prototypes by making the prototype assumptions better match the conditions under which the system will execute. By refining the prototypes, designers can explore the performance trade-offs of various alternatives with increased accuracy when needed to inform a design decision.

6.1.3 Design decisions

Component Selection

Often when building complex systems, designers may choose to use a third party component or library to implement part of the requirements. The I-Help discussion system uses a RDBMS to store persistent data and ensure ACID (atomicity, consistency, isolation, and durability) properties are maintained for concurrent requests. Many applications use RDBMS components to implement this functionality, but there are a number of options when choosing an RDBMS. While benchmark like TPC-C and TPC-H give general performance on large workloads [48], knowing how a given RDBMS performs on the target workload may be far more important.

To compare alternative RDBMS components for the I-Help system I have implemented a number of *StorageManager* classes that utilize different RDBMS components including: MySQL, MS SQL, Oracle, PostgreSQL, Sqlite, and flat files. While Sqlite and flat file are not full implementations since they do not guarantee transactional properties. Figure 6.2 gives a class diagram that shows how each prototype implements the *StorageManager* interface so we can easily change which prototype to use.

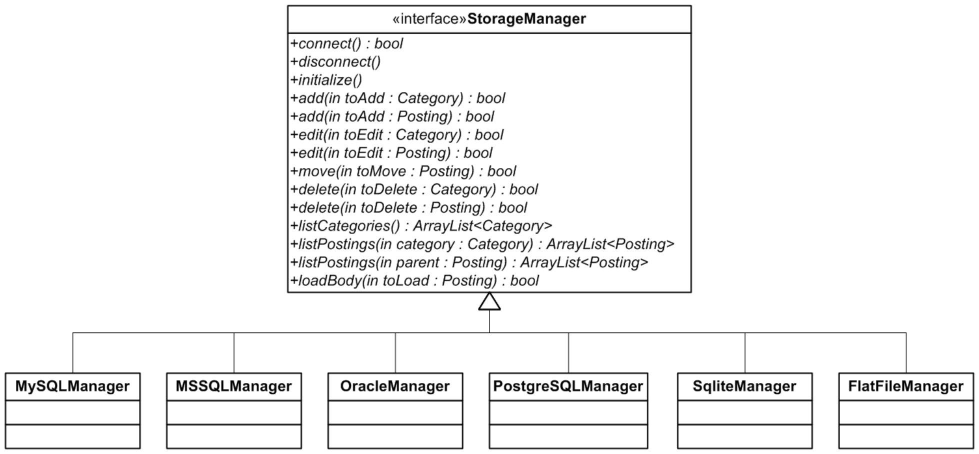

«interface»StorageManager
+connect() : bool
+disconnect()
+initialize()
+add(in toAdd : Category) : bool
+add(in toAdd : Posting) : bool
+edit(in toEdit : Category) : bool
+edit(in toEdit : Posting) : bool
+move(in toMove : Posting) : bool
+delete(in toDelete : Category) : bool
+delete(in toDelete : Posting) : bool
+listCategories() : ArrayList<Category>
+listPostings(in category : Category) : ArrayList<Posting>
+listPostings(in parent : Posting) : ArrayList<Posting>
+loadBody(in toLoad : Posting) : bool
MySQLManager
MSSQLManager
OracleManager
PostgreSQLManager
SqliteManager
FlatFileManager

Figure 6.2: *StorageManager* interface using MySQL, MS SQL, Oracle, PostgreSQL, Sqlite, and flat files to provide persistent storage for the I-Help prototype.

The source code of the Storage Component Test Case in Listing A.4 shows how one could compare each of these implementations. The test suite would instantiate a test run to collect the total running time for each of the implemented storage managers. While other metrics such as network traffic, CPU usage, disk usage, and memory consumption might also be important when comparing alternative storage mechanisms, for this experiment I have focused on total running time. Upon running this test suite, we can compare the total running time for each data storage component.

Data Caching

One way to improve response time is to cache popular persistent objects in random access memory (RAM) so as to avoid a disk access each time these objects are requested. Since cached objects are in memory we can also avoid the overhead of converting the persistent representation to an in memory object graph. Furthermore, in an n-tier architecture there is some overhead involved when sending queries from the application server to the database server, even in the case when both servers are running on the same physical machine.

Modern RDBMS systems generally use caching to achieve better performance, but often designers can achieve a greater performance improvement by building an application-specific caching mechanism in the application tier. To test whether application tier caching would be beneficial for the I-Help discussion system, I implemented a least recently used (LRU) cache system for caching the body of *Posting*s. Since some posts are likely far more popular than others, the cache will reduce the number of repeated database requests for the same, popular content.

To implement the LRU caching system for message bodies, I implemented the *StorageManager* interface shown in Figure 6.3. The *CacheManager* has an underlying data source (e.g. MySQL-Manager) that it retrieves data from, but when data is retrieved it is stored in the hash table named *m_ht*. Further requests for items that are found in the hash table are not forwarded to the underlying data source, they are simply retrieved from the hash table. The hash table has a fixed capacity and as such items may need to be replaced; to facilitate this replacement we track the last access time for each item in the hash table and replace the item least recently accessed in accordance with LRU. Since the maximum capacity of the cache hash table is configurable, we can experiment with a variety of cache sizes, trading RAM usage for reduced response times.

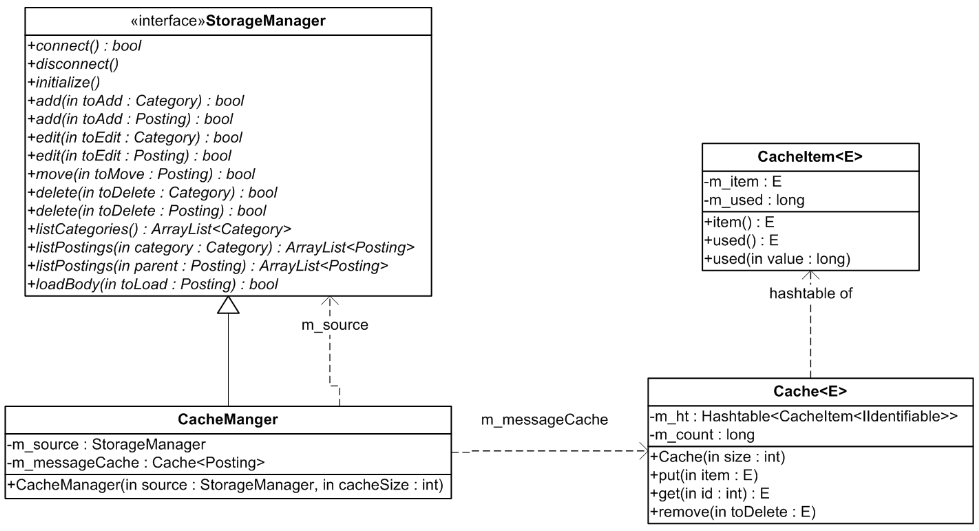

«interface»StorageManager
+connect() : bool
+disconnect()
+initialize()
+add(in toAdd : Category) : bool
+add(in toAdd : Posting) : bool
+edit(in toEdit : Category) : bool
+edit(in toEdit : Posting) : bool
+move(in toMove : Posting) : bool
+delete(in toDelete : Category) : bool
+delete(in toDelete : Posting) : bool
+listCategories() : ArrayList<Category>
+listPostings(in category : Category) : ArrayList<Posting>
+listPostings(in parent : Posting) : ArrayList<Posting>
+loadBody(in toLoad : Posting) : bool
m_source
CacheManger
-m_source : StorageManager
-m_messageCache : Cache<Posting>
+CacheManager(in source : StorageManager, in cacheSize : int)
m_messageCache
CacheItem<E>
-m_item : E
-m_used : long
+item() : E
+used() : E
+used(in value : long)
hashtable of
Cache<E>
-m_ht : Hashtable<CacheItem<IIdentifiable>>
-m_count : long
+Cache(in size : int)
+put(in item : E)
+get(in id : int) : E
+remove(in toDelete : E)

Figure 6.3: The caching prototype introduces a new storage manager that caches the messages returned by the underlying storage manager, thereby reducing the load on the underlying source.

The test case used for caching is the same as was used for component selection. The driver class which instantiates a test suite is shown in Listing A.6. This test suite compares a non-cached MySQLManager with two CacheManagers with cache capacities of 10 and 50 items.

Data Format

Rich web application interfaces have been facilitated by the spread of web browsers with support for asynchronous JavaScript and XML (AJAX). While these new interfaces provide clients with new types of interactivity, AJAX can also be used to reduce the amount of data transfered from a web server to the client since data can be processed and formatted on the client side. I-Help uses AJAX to provide an interactive user interface to its clients. Requests for data are made using JavaScript on the client to which the server responds with XML data. The XML response is formatted on the client-side and displayed to the user.

This design decision explores how a number of different data format options affect the total data transfered from the web server to the client when a given page is requested. Each format option is presented below:

1. Send a JavaScript formatting script along with XML data. This is the approach used in the current implementation of I-Help.
2. Send the entire file as static, formatted HTML document.
3. Send an XML stylesheet transformation (XSLT) document and XML data document.

While the data transfered is different in each of the above options, the resulting document displayed to the user in each case is identical. Additionally, since most web browsers and web servers support gzip compressed streams, I wanted to see how compression affects each of the above data transfer options.

The test case for this design decisions does not use the framework; it is an ad hoc test that fits within the process developed in Chapter 3. Because this test case is so simple, using the framework would not provide any benefit for this test. To execute this test case, I downloaded the XML data and JavaScript formatting script for the list of postings in a category produced by the current production implementation of the I-Help discussion system. I applied the JavaScript formatting script to the XML data to produce a static, formatted HTML file. Finally, I wrote an XSLT file that would transform the XML data provided into the HTML document produced by the JavaScript formatting script.

While this test case does not use the framework, it was extremely easy to implement, and gave interesting results which are discussed later in this chapter.

6.2 Design Decisions Experiments

In this set of experiments, I use design test suites to estimate the performance of alternatives for three real design decisions. Designers can take real design decisions and prototype them effectively using the DTPT process and framework.

With these experiments, I attempt to show that results obtained by executing a prototype approximates the results we would see in practice if we actually implemented each of the alternatives. While the framework will not produce 100% accurate results when executing against a prototype, it provides results that are consistent with those obtained from a full-scale implementation. This is important as it shows that for the situations tested, the prototyping approach does not mislead designers.

6.2.1 Procedure

In the first two experiments I used the basic prototype of the I-Help system introduced earlier in this chapter. This prototype is only 909 lines of Java code.

After completing the prototypes for each design decision, I began writing and running design test suites. The framework lets designers automatically execute test cases and collect performance metrics. After collecting performance metrics for each prototype, I compared the metric results to actual results obtained by instrumenting the production system in a similar way.

Each of the following subsections explains the details of the prototypes constructed, the performance results obtained, and an analysis of those results.

6.2.2 Data Caching

Prototype Details

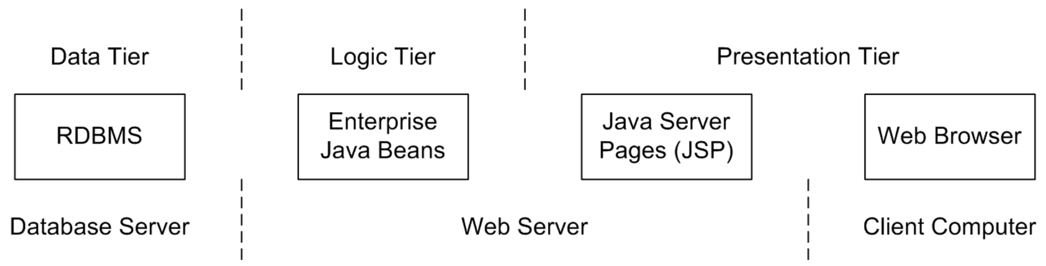


Figure 6.4: Multi-tiered architecture of the I-Help Discussion system.

I-Help uses a multi-tiered architecture shown in Figure 6.4, a common way to build and setup web-based Java applications. While multiple tiers introduces a good separation of concerns, it also introduces communication overhead between the tiers. In this example, we focus on the overhead of accessing and storing data on the database server, including: disk reads/writes, query compilation, converting data from a relational representation to an object representation, and network communication. While running both the database and web server on the same physical machine can reduce/eliminate network overhead, there is still inter-process communication (IPC) overhead between the two.

One way to reduce the overhead of accessing data is to cache commonly accessed data items. Most storage components (e.g. an RDBMS) cache queries and the results of those queries, but this only eliminates the disk read/write and query compilation overhead. Caching data in the logic tier can save us all the overhead of accessing the data tier at the expense of adding complexity and storage to the logic tier. We can reduce the complexity and storage requirement by caching only the most popular subset of data items in the logic tier and continue to access less frequent items via the data tier.

When caching data that can be modified during application execution we must ensure the data in the cache is updated when it is modified. While there a number of ways to ensure this, the prototype I have developed does not implement this behaviour.

In the prototype, I was hoping to understand the effectiveness of this type of caching system without integrating it into the I-Help system. To accomplish this, I built a simple LRU cache class, a class to generate a stream of requests with a Zipf distribution, and a design test suite driver class.

The Zipf distribution shown in Equation 6.1 describes the popularity of the k-th document in a collection of N documents when in ascending order by rank [1]. The Zipf distribution used in this prototype sets the exponent shape parameter s as 1.

$$f(k; s, N) = \frac{1/k^s}{\sum_{n=1}^{N}(1/n^s)} \tag{6.1}$$

A stream of requests is simply an array of integers each corresponding to the ID of the document being requested. The design test suite runs three iterations, then calculates the average cache hit ratio when varying three parameters:

- **Cache size** - the number of discussion posts that can be stored in the logic tier cache
- **Number of Documents** - the number of unique discussion posts in the request stream
- **Number of Requests** - the length of the request stream

Table 6.1: Cache hit ratios found in the prototype assuming a Zipf distribution.

Cache Size	No. of Documents; No. of Requests		
	100; 5,000	200; 5,000	2,000; 10,000
Prototype Cache Hit Ratio (%)			
0	0	0	0
30	0.6154	0.4858	0.2843
50	0.7584	0.6066	0.3562
80	0.9178	0.7256	0.4223

Results and Analysis

The results in Table 6.1 shows that for a small cache of 30 documents and a collection of 2,000 documents while the cache contains only 1.5% of the collection at a time, the cache hit ratio is 28.43%. This means that even a small cache can significantly reduce the number of documents that must be fetched from the data tier.

The cache hit ratio is high due to our assumption that document popularities follow a Zipf-like distribution. To test how well the prototype predicts the benefit found in an actual implementation I produced a working implementation of this caching system.

In the full implementation the data tier consists of a MySQL RDBMS running on the same machine as the logic tier. I populated the database with postings scraped from a popular online discussion forum [1], and replaced the Zipf distribution of popularities with the actual posting popularities scraped from that site. The cache uses LRU to decide which documents to keep in memory and falls back to a Java Database Connectivity (JDBC) data access layer to load data from MySQL. The results shown in Table 6.2 show the actual cache hit ratio and total running times averaged across three runs of the experiment.

Table 6.2: Shows the hit ratio and total running time to service an actual request stream.

Cache Size	No. of Documents; No. of Requests		
	100; 5,000	200; 5,000	2,000; 10,000
Actual Cache Hit Ratio (%)			
0	0	0	0
30	0.6474	0.5000	0.3443
50	0.8240	0.6846	0.4774
80	0.9562	0.8244	0.5989
Actual Running Time (sec)			
0	5.344	4.938	9.719
30	1.844	2.578	6.563
50	0.969	1.703	5.422
80	0.265	0.984	4.265

Table 6.2 shows that the prototype accurately predicted the results found in practice. The cache hit ratio obtained by each cache size is slightly higher than those hit ratios found in the prototype (suggesting actual probabilities are less heavy tailed than a Zipf distribution), and total running time reduced by 32% by introducing a 30-document cache for the collection of 2,000 documents.

Designers facing the decision of implementing an logic tier cache can quickly write a prototype and use the framework to predict the cache hit ratio. While the prototype implemented does not predict the cache ratios exactly, it does show the same behavior across the set of parameter values as the full implementation. Thus, this experiment shows the value of using prototypes with the framework to predicting performance metrics of real design decisions.

[1]Postings and popularities were scraped from http://www.sitepointforums.com, a popular forum for web developers.

6.2.3 Component Selection

When building a large-scale application, using existing components can significantly reduce the total time required to implement and test the production system. Furthermore, many efficient and robust components are widely available [12]. Relational Database Management Systems (RDBMS) are a classic example of well tested, efficient middle-ware components that many developers use in production systems. There are a large number of open-source and commercial multi-purpose RDBMS available. The I-Help discussion system uses a MySQL, a RDBMS, to persist data in the data tier.

While a number of different RDBMS exist, choosing one can be a difficult task since some developers have strong preferences and opinions. After deciding on options that fit the application requirements and budget, performance prototyping seems like a great way to compare components early in the application design.

Prototype Details

In this experiment, I have installed a number of RDBMS systems to compare how these components perform given a realistic workload. While organizations like the Transaction Processing Council (TPC) [2] have established industry-standard database performance benchmarks, many applications may use only a small subset of the features tested in these benchmarks and have access to significantly less advanced hardware than those used to run the benchmarks. These experiments exercise a workload that is representative of the application under consideration; thus the results predict performance metrics of each RDBMS specifically for the application under consideration on whichever hardware platform the designer wishes to run the tests.

In my prototype experiments, all of the databases used were installed and run ‘out of the box’, without any tuning. All databases were installed on the same physical machine that runs the design test suites and all connections are established via a TCP/IP connection to localhost (named pipes/IPC were not tested). Furthermore, all of the database servers were running for the entire experiment running time. While this means each server would compete for resources, the tests can be run very quickly (without starting and stopping services), and we do not have to account for the overhead of starting the database server in each test case.

The prototype consists of a single JDBC data access class which issues SQL statements to an underlying JDBC driver. The test case uses the actual request stream generator created in the last set of experiments, but requests are serviced by a multi-threaded server (to model how an HTTP server would service requests). The test case starts one writer thread which makes edits to documents in the database while the other threads read documents from the database (each thread

[2]For more information about the TPC, see: http://www.tpc.org/

uses a separate JDBC connection). Together, all the threads make a total of 10,000 edits/requests.

Since SQLite3 is not a true client/server RDBMS, it does not connect via a TCP/IP connection, but rather uses the Java Native Interface (JNI) to convert JDBC calls to SQLite3 library function calls [3]. Furthermore, the Flat file component shown below is shown only for reference as it is not an RDBMS, but simply reads postings from and writes postings to files. The Flat file component uses Java monitors to ensure only one thread at a time accesses a given post.

Table 6.3 shows the total running time and heap memory usage for each RDBMS over 10,000 requests distributed amongst two, six, and ten parallel threads. The reported heap memory usage does not include memory used by the RDBMS server process, only for the test suite and JDBC driver.

Table 6.3: Total time required and heap memory overhead to service request streams using different numbers of threads for various RDBMS components.

Threads	2		6		10	
	Time (sec)	Memory (kb)	Time (sec)	Memory (kb)	Time (sec)	Memory (kb)
MySQL 5.0 MyISAM	12.078	2,144	7.578	3,436	7.078	3,218
MS SQL Server 2005 Express	26.906	3,148	15.063	3,357	14.016	3,212
Oracle 10g Express Edition	51.265	3,082	44.078	3,229	38.14	2,973
PostgreSQL 8.3	31.187	3,366	19.14	3,639	15.75	3,854
SQLite3	362.766	10,527	202.359	14,567	197.703	17,655
Flat file	43.891	13,720	12.312	7,607	8.141	4,196

Table 6.4: Total time (in seconds) required to service a steam of requests in the actual system.

Threads	2	6	10
MySQL 5.0 MyISAM	1.625	1.719	1.162
MS SQL Server 2005 Express	5.078	2.694	2.994
Oracle 10g Express Edition	21.141	41.750	17.522
PostgreSQL 8.3	2.062	2.134	3.303
SQLite3	20.009	12.350	9.556

Results and Analysis

The Indexed Sequential Access Method (ISAM) used by MySQL for storing data is ideal for fast retrieval. This is likely the cause of MySQL's excellent performance on this workload, which uses

[3]For more information on the SQLite Native JDBC Driver, see http://www.zentus.com/sqlitejdbc/

neither an advanced configuration (clustering/replication) nor advanced SQL commands (transactions, rollbacks, joins).

All of the components complete the workload significantly faster when six threads run in parallel rather than only two. While ten threads provides some benefit it is not as pronounced as moving from two to six.

SQLite3 performs very poorly on this work load. The SQLite database speed comparison has similar results when full disk synchronization is turned on [4]. Unfortunately, the JDBC driver does not document how to toggle synchronization on or off. Furthermore, this JDBC driver uses the most heap memory, likely since all the work is happening inside the driver and its native JNI agent, while the other RDBMS systems simply dispatch requests to the server process over TCP/IP.

The flat file implementation is rather simple, but performs very well. This workload only looks up postings based on the primary key, which corresponds to the filename in the flat file component. Thus, if the workload involved index lookups on a non-primary key column (a common situation), the flat file component would be much slower.

To ensure the results obtained by the prototype are consistent with those obtained from the production system, I ran a second experiment. In this experiment I executed a workload in the production version of I-Help and logged the queries sent to the database component. The average time to execute the workload for each system is presented in Table 6.4. This table shows the performance for 2, 6, and 10 threads concurrently executing queries in the workload.

While changing the database component may seem like a small change to make, this change involved significant design, development, and testing to execute. While each of the database components adhere to portions of the SQL standard, they vary in a number of ways from the datatypes they accept and the enforcement of foreign key constraints, to text escaping and built in functions. Building a prototype system to compare these components was significantly easier than modifying the existing software system to test the same hypothesis. Modifying the production system took an order of magnitude more work than building the prototype.

Oracle also performs poorly on the actual and prototype workloads. Oracle and to some degree Microsoft SQL Server are both optimized for running complex queries on large databases. The system we have implemented is mostly running simple SELECT and UPDATE queries based on the primary key of very small tables (under 5 mb). In the full-scale implementation Oracle performs very poorly with six threads as compared to two or ten threads. There is some behaviour occuring (consistently under multiple test runs) that causes this performance problem that is not reflected in the prototype. The prototype is a simplification of the actual implementation, and thus does not predict this odd behaviour.

PostgreSQL performs progressively worse as the number of concurrent threads is increased in the

[4]see http://www.sqlite.org/speed.html for more details

production system, yet this behaviour is not exhibited by the prototype. The prototype does not implement transactions, and does not maintain a history of changes while the production system does both. The course-grained data locking and maintaining the history of every posting update seems to cause poor performance for the PostgreSQL database backend in the production system. Likely this problem could be alleviated in production by optimizing the write locks.

Because of these odd behaviours not exhibited by the prototype, I revisted the prototype to see if any of the simplifications seemed unreasonable. Instead of running an actual webserver and simulating actual requests, I just spawn a number of threads and those threads execute actions directly. While this simplification wouldn't affect database performance significantly, the way in which I originally implemented seemed unrealistic upon further examination. In each of the test cases, there is a single writer thread, and some number of reader threads. This does not match how the application would perform in practice, since each thread handling web requests would have a similar probability of reading or writing to the database for each request.

I refined the prototype such that each request, distributed across all threads, would have a 15% probability of writing to the database, rather than having a single writer thread. I re-ran the same experiments with this new prototype and obtained the results shown in Figure 6.5. To better understand the trends occuring I executed the prototype for each number of threads from one to ten. The results obtained from the refined prototype match those found in the actual system much better:

- PostgreSQL performance degrades slightly as more threads are added.
- Oracle exhibits worse performance than SQLite over the experiment.
- MySQL, MS SQL, and PostgreSQL perform significantly better than SQLite and Oracle.
- As the number of threads is increased, Oracle's performance is worst at a moderate number of threads.

The first experiment shows that we can quickly compare components using a simple prototype, and the second prototype shows that the results obtained from a simple prototype are consistent with those obtained from the production system. While the prototype was fast to construct and yielded good performance estimates, it did not produce the exact same results as the same tests in the actual system. Furthermore, I have shown how designers can revisit prototypes to fix poor assumptions and refine the prototype to acheive more accurate performance estimates.

6.2.4 Data Delivery Format

The I-Help discussion system is a web-based application and total data transfered is an important metric both from a cost (since bandwidth costs money) and client satisfaction (since more data means longer load times) point of view.

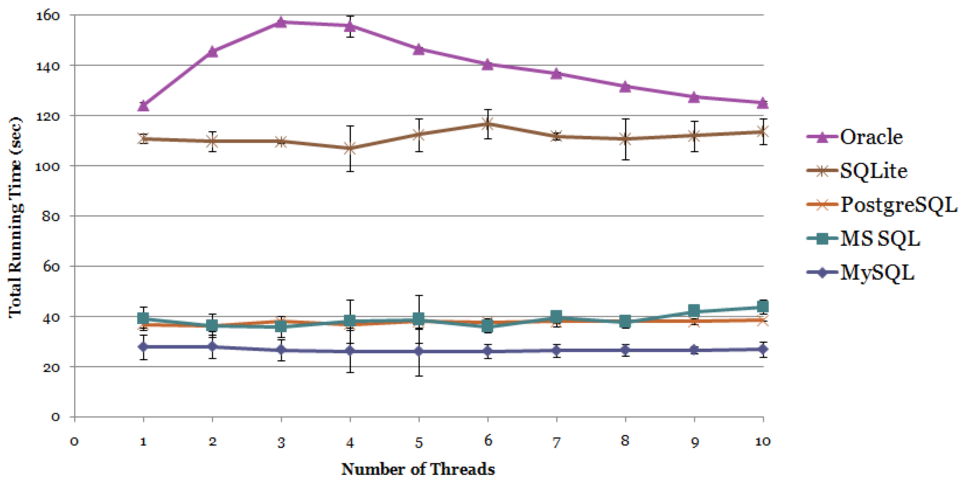


Figure 6.5: Results for the database comparison after refining the prototype.

With computers becoming faster and client-side scripting languages like JavaScript maturing to be more compatible across platforms, web applications commonly push some processing to the client side. Asynchronous JavaScript and XML (AJAX) and Extensible Stylesheet Language Translation (XSLT) are two methods that allow servers to transfer raw data to clients and have those clients format that data for display as Hypertext Markup Language (HTML) and Cascading Stylesheets (CSS).

In this experiment I consider the question: can I-Help use these new methods to reduce the total data transfered from the server to a client?

Prototype Details

This experiment does not run within the DTPT Framework, but is an ad hoc test suite that follows the process introduced in Chapter 3. Because this prototype test involves simply measuring the filesize of data in different formats, the framework would provide little benefit for this test. The prototype consists of four files:

1. The XML data that lists all the discussion posts.
2. A formatted HTML document listing the discussion posts.
3. A JavaScript document that requests the raw XML data and formats it to look like the HTML document.
4. An XSLT document that transforms the raw XML data into the formatted HTML document.

The prototype experiment compares the total bytes transfered for each of three data formats: AJAX (Javascript + XML), Transformation (XSLT + XML), and Formatted HTML (HTML only).

Since most clients support some form of HTTP compression we also compared the size of these files after compression with G-Zip (a popular HTTP compression library).

The actual I-Help discussion system transfers data using AJAX, so to compare the prototype results to results from a full implementation, I downloaded the actual XML and JavaScript files from an I-Help discussion thread. I then wrote an XSLT file that produces the same formatted data as the JavaScript file given the XML input. Finally I ran the XSLT file on the XML input to produce a static HTML document.

Table 6.5: Shows the data transfered for the first and second response using various data formats.

	Prototype (bytes)		Actual (bytes)	
	Raw File	G-Zip Compressed	Raw File	G-Zip Compressed
Individual File Sizes				
XML Data	2,315	1,145	39,643	5,259
HTML	2,119	1,159	30,839	4,759
JavaScript	1,183	533	40,526	9,738
XSLT	552	327	2,831	1,121
Option 1. AJAX				
First	3,498	1,678	80,169	14,997
Second	2,315	1,145	39,643	5,259
Option 2. Transformation				
First	2,867	1,472	42,474	6,380
Second	2,315	1,145	39,643	5,259
Option 3. Formatted HTML				
First	2,119	1,159	30,839	4,759
Second	2,119	1,159	30,839	4,759

Results and Analysis

The results in Table 6.5 show clearly that ASCII data like HTML, XML, XSLT, and JavaScript benefit significantly in the prototype and actual implementation from G-Zip compression. Furthermore, the first and second options (AJAX and Transformation) both use less data for the second request (since the display processing information need not be transfered again), while Formatted HTML requires the same bandwidth for each request (since the formatting and data are integrated in the same file).

Both the prototype and the actual implementation exhibit the surprising result that a single formatted HTML file requires less bandwidth to transfer than the raw XML data; not including the JavaScript or XSLT formatting code and HTTP overhead of requesting multiple files. While this result is surprising, it can be explained:

1. Some of the data in the XML is used in processing the file, but not displayed in the resulting output (e.g. which posts have been seen previously).

2. The XML format is bulky for encoding repetitive data, a format like comma separated values (CSV) or JavaScript Object Notation (JSON) [5], could significantly reduce the data size (but would not allow the XSLT approach).

While the formatted HTML transfer format uses the least bandwidth, the application designers may not choose this format. First, since clicking on any link requires a full page refresh, rather than partial page update as can be accomplished using AJAX, which may make the client think the application is slower even though it is not. Second, designers may wish to push the computation required to format the data from the server to the client, reducing the total server load.

6.3 Interesting Discoveries

While building the prototypes and design test suites for the I-Help discussion system design decisions I made two interesting performance-related discoveries. Since I made these discoveries while implementing the prototypes, it would be possible for me to incorporate these lessons in to the system design prior to starting the system implementation.

The first working versions of the JDBC and Flat file persistence classes were using a large amount of heap memory. When loading a document with the *loadPosting* method, these classes return a new instance of the *Posting* class. Since the design test suite loads up to 10,000 postings per iteration, these classes were instantiating a lot of *Posting* objects. To reduce the heap memory usage, I created a *PostingPool* class that instantiates a pool of *Posting* objects initially, then gives these out and the application returns them to the pool when it is done with them. This is similar to explicit memory management accomplished using *malloc* and *free* in C.

The idea of an object pool makes a lot of sense for this application since postings are loaded, sent to the client, and discarded. Even when we introduce a cache, the cache contains a fixed number of postings and thus the object pool can still be used and remains a constant size. Not only did this reduce the total heap memory usage, but it also reduced the total running time since the Java Virtual Machine had less heap allocation and garbage collection to do. This is also a substantial improvement since the object pool itself is only 45 lines of Java code, and required only four code changes to integrate into the prototype.

[5] See http://www.json.org/ for more details.

When running the RDBMS prototypes I had thought there would be significant overhead involved in running the first query to the RDBMS (paging the database server back into memory, initiating the buffer pools, etc.), so I had the test suite execute the first test case for a database twice so I could compare the result of the first and second run. Surprisingly, the results were nearly identical for the first and second run of the test cases for all of the databases. My intuition that RDBMS would have a significant cold start cost was not correct for this workload.

6.4 Conclusion

Through a series of experiments I have attempted to show the usefulness and validity of the process and framework for design-time performance testing. The experiments in this chapter explored the previously introduced process and framework and the results show that:

1. Prototypes can be written with very little code and still provide meaningful results. The entire I-Help system was prototyped in less than 1,000 lines of Java code.

2. Designers can use the framework to prototype design decisions, gather performance metrics, and use the results to make design decisions. While the prototypes could not predict system performance exactly, the performance results for the prototypes accurately predicted the relative performance seen in full implementations.

3. Prototypes can expose interesting, unexpected performance behavior early in the development process, allowing designers to use these results to inform their design decisions.

Chapter 7

Discussion

This discussion focuses on how the process and framework presented in this thesis relates to work in the field of software development. I start by considering some basic questions about the key parts of the framework, and how developers can effectively use the framework. Next, I look at the importance of documenting performance results and how the framework automates this task. I then show that using the framework to test a number of different situations requires minimal modification of the prototypes and test suites. Finally, I show real world applications the process and framework have been applied to and suggest others to which I believe it is well suited.

7.1 Framework

The framework implementation strives to not only achieve the goals laid out by the process, but to also allow practitioners to very easily integrate its use into their development practices. My approach was to structure the framework around familiar constructs including test suites, performance metrics, and software prototypes, to make it fit seamlessly into the software development tool chain.

7.1.1 Test Suites

jUnit is a unit testing framework that is widely used in industry [34]. To build on the success of jUnit, the framework in my thesis builds on a number of concepts:

1. Test cases and test suites are written in Java. Since test cases are written in the same language as the final implementation, developers can draw on their experience in and knowledge of Java to write test cases by learning only a few special directives to the testing Framework.

2. Test suites consist of a collection of test cases. These collections allow developers to easily run a collection of tests and look at the aggregate result.

3. Test cases can be executed automatically from the command line, and results can be presented in a human-readable or graphical way.

While using test suites gives us a number of benefits, it also raises a number of important questions:

1. What is a *good* test case?
2. On which situations should designers focus?
3. Are there techniques we can use to reduce the overhead in metric results?

A *good* test case is therefore one that provides accurate and timely insight to designers regarding a specific design decision. Test cases should aim to answer performance questions specific to the application under test. Furthermore, since writing test cases has an opportunity cost, good test cases should also be quick to develop, relative to the implementation whose performance they are estimating.

Design test cases provide performance metric estimations that can be used by designers to compare alternative implementations. Therefore, designers should focus on those situations which the performance metric results will influence their decision. This includes situations where some of the alternative have significant performance benefits or drawbacks, and those where the decision must be fixed early but the performance of alternatives is not well known. Building prototypes and running test cases can also be used to diffuse a religious debate within a design team.

Design test cases are not intended to be used for optimization or fine tuning. To reduce the overhead of collecting metrics, designers should focus on testing situations (rather than individual operations). The framework introduces both a fixed and variable overhead when collecting performance results. By testing situations rather than individual operations, we can reduce the effects of the fixed overhead. Designers can adjust the active metrics to include only those which provide insight, and change the sampling frequency to reduce the variable overhead.

7.1.2 Metrics

The process and framework use metrics to compare the relative performance of alternatives. The framework includes a number of standard metrics and also allows developers to build custom metrics by extending framework classes. A good metric is one that concisely summarizes the important characteristics of an application. By allowing developers to extend the set of metrics provided, I have allowed developers to define new metrics that may be better suited to their specific applications.

While metric values can be graphed or displayed in a table automatically, understanding the meaning of metric values can be complicated. For example: in a situation where caching can reduce total running time at the cost of increased memory usage, a designer must decide whether to employ caching, and how large to make the cache. While performance metrics will help the designer, the meaning of those metrics must be understood in the context of the application under test, including:

- What memory is available?
- What is a reasonable response time?
- What are the consequences of a slow (cache-miss) or an out-of-date (cache-hit) response?

7.1.3 Prototypes

My approach to design-time performance testing relies on using a prototype to approximate the performance behavior of a production implementation. Since prototypes are so important to the approach, it is worth considering how to write good prototypes. Since the purpose of a prototype is to provide an accurate performance estimate, it is not important that the prototype provides a complete or correct solution, only that it has the same performance characteristics as a complete or correct solution. For example, when prototyping an application which counts the number of leaf nodes in a tree, it not important that the number of nodes is reported correctly; it is only important that the prototype behaves like the desired tree exploration algorithm.

7.2 Documenting Results

While obtaining accurate performance metric estimates is fundamental to my work, presenting those metrics to designers so they can easily compare alternatives and make decisions is also important. Ward Cunningham's Framework for Integrated Test (FIT) takes a document containing test case parameters and their expected results, then interleaves the results of executing those test cases into the documents [19]. Cunningham's framework makes reviewing test cases and understanding the results of their execution extremely easy even for stakeholders not familiar with writing code or test cases.

The framework I have developed includes two mechanisms for documenting the results of executing a design test suite. First, performance metric results can be graphed using an export to gnuPlot; this can be very useful for understanding how parameters affect a set of performance metrics across multiple prototypes (e.g. the sorting comparison in figures 5.2 and 5.3). Second, I have implemented a replacement engine that copies metric results into formatted documentation based on variables embedded in the document. This can be used to achieve similar results to the FIT framework and is compatible with a variety of document formats including: Plain Text, HTML, and LaTeX. Figure 7.1 shows an example HTML document with variables embedded in curly braces and Figure 7.2 illustrates the corresponding output generated. These will be replaced by the parameter and metric values from an actual test run by the replacement results formatter.

Both of the tools for documenting test results can be integrated into a design test suite so generating documentation is an automated part of executing design test cases. Users of the framework can also write their own metric formatters to generate other types of visualizations.

```
 <html>
<body>
<p>For an input size of <b>{param[1][1]}</b> items and a cache of size
<b>{param[2][1]}</b> items took <b>{TotalRunningTime[1][1]}</b>.</p>
<p>For an input size of <b>{param[1][2]}</b> items and a cache of size
<b>{param[2][1]}</b> items took <b>{TotalRunningTime[2][1]}</b>.</p>
</body>
</html>
```

Figure 7.1: Example replacement HTML file. The results formatter replaces the values in curly braces with the metric and parameter values produced by running the test case.

For an input size of **100** items and a cache of size **8** items took **14.3 seconds**.

For an input size of **100** items and a cache of size **16** items took **9.6 seconds**.

Figure 7.2: Example result of executing a test case and formatting the result with the replacement engine.

7.3 Adaptability

Test cases and prototypes built for the process and framework can be easily adapted. This is important since they are intended to be used early in the development cycle, at which point the system requirements and design ideas are often volatile. This also allows designers to quickly prototype and test the performance of new ideas. Because prototypes are only approximations of a full-scale implementation, testing a potential design change in a prototype should require far less involvement than in a full implementation.

For example, let us consider a few ways designers might wish to modify the first I-Help experiment in the previous chapter where we are comparing relational database management systems:

1. If *transactional updates* were a known bottleneck, we could prototype this by adding a transactional update method to the storage manager classes (JDBC and FlatFile), then simply add that action to our work load. After these minor code changes we could collect new performance metrics using the same design test suite with the modified workload.

2. While I-Help is written using a *tiered architecture*, all three server-side tiers run on a single physical machine. If we wanted to consider separating the data tier to its own server from the logic and presentation tiers, we would simply need to execute the design test suite from

a different machine, modify the connection strings used in the JDBC storage manager, and map a network drive for the FlatFile manager.

3. We can easily compare different database communication protocols and JDBC drivers by changing the connection strings in the JDBC storage manager and installing further JDBC drivers.

4. If we are concerned about the overhead introduced by running all RDBMS systems concurrently, we would simply need to modify the test case to startup/shutdown each RDBMS server in their setup/finish methods.

While this is only a small sample of potential adaptations, it does illustrate the ease of adapting test cases and prototypes for some situations.

7.3.1 Design is Too Early for Performance

Some authors argue that performance is best left as a final step when developing software. Harold shares this point of view in his article *Correct, Beautiful, Fast (In that order): Lessons From Designing XML Verifiers*, from the book *Beautiful Code* [23]. Those with this perspective may perceive my work on trying to get designers to consider performance earlier as counter-productive. It is important to remember that design-time performance testing does not preclude the notion that correctness, maintainability, and readability may all be more important than performance, in fact they usually are. Design-time performance testing simply intends to inform designers by helping them understand the performance differences of various alternatives, they can then prioritizes these trade-offs with the other design qualities of each alternative.

It could be further argued that giving designers and developers tools to quickly compare performance options may distract them from their core task of developing or designing the software system. While this is a possibility, it is not the intent of my approach. I hope designers and developers can quickly test alternatives, use the information gathered using the framework and/or process to inform their decision and then continue working on the task at hand. Because the tool uses prototypes to approximate full scale implementations it should not be used to fine tune or tweak early in development, but rather to quickly explore potential solutions. For the task of optimization and fine tuning, developers should continue to use tools like profilers and optimizers late in the development cycle, since at that point the rest of the system is stable, allowing for small tweaks and optimizations.

7.4 Applications

The value of a process and framework for informing design-time performance testing is not only an academic exercise. Ideally, I hope designers and developers can use and build on this work to help them design and implement practical software systems. To show that this is possible, I have applied the process and framework to two production software systems and also outline how it might be used in other scenarios.

7.4.1 I-Help Discussion

The I-Help Discussion system is a three-tiered production software application written in Java. A number of my experiments show how designers can use the framework and ad hoc tests to compare performance consequences of making alternate design decisions in accordance with the process introduced in Chapter 3.

While the prototype of the I-Help system was around 1,000 lines of Java code, we were able to test a number of scenarios with this prototype including five relational database management systems (RDBMS). Furthermore, this prototype could easily be modified to test different implementation ideas and new scenarios. The I-Help discussion system originally used Oracle to store and manage data, but has been running on MySQL for years now. While the development team has tried to restrict the system to using standard SQL, significant modifications were required to get I-Help running on Oracle and other RDBMS. Since the prototype is much simpler, getting it running on the various RDBMS was trivial in comparison.

To get I-Help running on my machine and get familiar with the system, I met with Colleen Hansen, a developer on the I-Help project. In my discussion with Colleen, I found out that scalability is currently an important issue for the I-Help development team. The system was used by four colleges and about 20 classes in the Spring 2008 semester, but will soon be integrated into PAWS, the student access system for the entire university. To prepare for the increased load this integration presents, the developers are rewriting some of the queries which seem to be running the slowest. The process and framework could assist the I-Help development team in speeding these queries up.

While tools like Microsoft's SQL Query Analyzer are able to dissect a query into individual operations and tell users which portions of the query are taking the longest, the results of these tools are not easy to understand. By copying the original query into a prototype as well as prototyping a few new alternatives (possibly splitting the query up into smaller queries, or adding an index), users can quickly compare how new alternatives affect the total query running time without having to use specialized tools or have an in-depth understanding of SQL query optimization. Furthermore, the process and framework can execute these tests against production data (or a copy of production data), so users are getting a very good approximation of how these changes will affect their application.

7.4.2 PHP Application

During the course of developing and testing the framework, I was experiencing an intermittent performance problem in a web-based, data-driven PHP (Hypertext Preprocessor) application which I had written for a client. While I was unable to reproduce the problem, the client reported it was happening at what seemed random intervals and often enough to be a major problem. Since the customer was not a software tester or developer, I was unable to get enough information about the problem to diagnose it fully.

To get a better understanding of the performance problem, I wrote an ad hoc test suite based on the process in my thesis that tested a number of possible system bottlenecks and reported the results as a formatted HTML document. I provided this application to the client so he could run it whenever the performance problem occurred. Only a few hours after sending the performance test suite to the client, I had received test results from the client. These test results showed me exactly what component was performing poorly, thus significantly reducing the number of potential solutions I might need to try to fix this problem. I was able to find the source of the problem within minutes: MySQL was taking three minutes to execute a summary query. While database performance can sometimes be fixed by tuning database paramters, in this case the data was being calculated in a way that was very wasteful. After finding the problem, I implemented and tested an alternative design which gave the same result, but performed much better. The solution was to break the query into three smaller queries which together executed in milliseconds. I was able to find and correct this problem based on the insight provided by the test results gathered.

7.4.3 Software Design

The primary use of this work is during software design. Design-time performance testing seeks to inform design decisions with performance information. Understanding the performance consequences of alternative implementations helps designers compare alternatives and brings performance considerations to light earlier in development (preventing "house on fire", late-stage performance tuning). Furthermore, using the DTPT process or framework formalizes the performance concerns of an application in the form of design test suites. While designers can choose the best performance alternatives, being informed also allows designers to choose alternatives that trade performance for other more important qualities (e.g. reduced development time), since they can accurately predict the cost of this trade off.

7.4.4 Component Selection

Composite systems built using existing open source and commercial components are becoming a popular way to build production systems to meet the ever increasing demands of application consumers. REST, SOAP, and XML-RPC based web services have created new opportunities for developers to quickly mash up existing systems in new and interesting ways. When selecting a third party component like a relational database management system (RDBMS), or a web service provider, the performance of that component may significantly influence the overall system performance. Additionally, third party components generally do not provide the same interface, so switching components is costly. Therefore, selecting a component with poor performance can cripple the resulting system and be difficult to change for an alternative component. Understanding component performance early is important.

The process and framework can be used to quickly prototype potential operations on a third-party component and compare alternative components for a number of application-specific scenarios. Benchmarks like those provided by the transaction processing council (TPC) provide certified industry standard performance results on a complex workload for database management systems. Alternatively, or in combination with these benchmarks, the framework can be used to quickly test how a given component will work within a given environment on an application specific workload. This lets designers quickly predict how components will behave in their application specifically.

7.4.5 Regression Testing

When upgrading from one version of a third-party component to another, application developers rely on the behavior of the previous version. If that behavior changes, developers need to modify their systems accordingly. While unit tests can be used to ensure a new component functions correctly, the design-time performance testing framework can be used to compare the performance of important scenarios on the new and old version of the component. This can help developers identify new performance problems, avoid bloated releases, and make the decision to switch to a faster or leaner competing component.

7.5 Conclusion

The framework in this thesis builds on concepts introduced by other testing frameworks like jUnit and the Framework for Integrated Test (FIT). The process and framework have been applied to two production applications and are sufficiently adaptable to quickly test a number of variations on those experiments performed in the previous section. I also highlighted three areas I believe the DTPT process and framework could be used effectively in practice.

Chapter 8
Conclusion

In this chapter, I restate the primary and secondary contributions of my thesis and highlight specific chapters in which each contribution is advanced. This chapter goes on to suggest possible future work and finally conclude the thesis.

8.1 Overview

Predicting the performance of even a simple software application at design-time is difficult because modern computers and languages are complex. Understanding the performance consequences of design decisions when writing a composite system built from 3rd party components deployed over a scalable array of computers adds a multitude of complexities to this type of performance analysis. Understanding and predicting software performance is becoming far more difficult as applications and their execution environments become more complex.

With Amazon EC2, Microsoft Live Mesh, and Google AppEngine competing to offer low cost, easily accessible cloud computing, building large scale applications that run on complicated architectures is no longer the exclusive domain of academia and big industry [1]. Small startups can scale their computing power on-demand without raising upfront capital, maintaining servers, or housing a server farm. Combining these low cost computing platforms with web services (e.g. Google Search engine, Youtube's Video library) allows startups to quickly create new composite software applications.

Performance is important for composite applications; Yahoo's Developer network has an entire section of articles with general performance rules for how to improving the performance of applications that use their web services [2].

Understanding important software qualities early in the design process helps software designers make design decisions. In this work I have considered how software designers can understand the performance (called efficiency in ISO 9126) of their designs to help them decide between design alternatives. I found from previous work that software designers largely rely on their past experi-

[1]Behind Live Mesh: How we run cloud services. http://blogs.msdn.com/livemesh/archive/2008/04/30/behind-live-mesh-how-we-run-cloud-services.aspx

[2]Exceptional Performance. http://developer.yahoo.com/performance/

ence, broad performance rules, and late stage optimization to deal with the performance of their software designs.

My thesis introduces a new approach to design-time performance testing: comparing the performance characteristics of a design alternatives early to help make design decisions. I developed a process which uses design test suites to approximate important design scenarios, prototypes to estimate alternative's to a given design decision, and gauges the relative performance of those alternatives by running them through the design test suites and collecting performance metrics.

While designers could use existing tools meant for unit (e.g. jUnit) or load testing to make these performance estimations, the process helps designers identify the inputs which will influence the approximate performance metric results. Through the experiments in my thesis I have shown that this technique provides valuable feedback to the design process. This work differs from traditional correctness and performance testing in that it is not intended for use on a single, fully featured implementation, but rather is intended for comparison of prototypes and partial implementations.

To show how the DTPT process could be applied to real situations I wrote the DTPT Framework for design-time performance testing that implements the process.

8.2 Contributions

8.2.1 Primary

The primary contribution of this work is a method of predicting performance characteristics using prototypes and test suites. In Chapter 3, I introduced a process of design-time performance testing that approximates alternatives with prototypes and scenarios with design test suites to calculate approximate performance metrics. This process provides software designers a formal way to understand and compare the performance consequences of various alternatives when making design decisions.

In Chapter 5, I verified that the process gives results that are consistent with our intuition and previous work.

I also showed in Chapter 6 that although the prototypes do not yield the same performance metric results as a full-scale implementation, they lead to decisions that are consistent with those a designer would make given performance results obtained from the full-scale implementation.

Furthermore, I explored how the process can be applied to the real world in Chapter 7. The process is intended to inform software design decisions, both when writing software or selecting a third party component. The process can also be used for documenting software performance results and application-specific regression testing. I applied the process to hypothetical design decisions in I-Help and a performance bottleneck in a web-based PHP application to show that it can be used on real world, production systems.

8.2.2 Secondary

1. I developed a framework implementation that allows developers to easily integrate the DTPT process into their development routine. This framework is similar to JUnit and is introduced in Chapter 4. The framework is written in Java with a native, portable C++ agent library. The framework makes design test suites and prototypes easy to express and formalizes them as Java classes that can be reused and evolved as needed. The framework includes:

    - a set of classes for running test suites and instrumenting prototypes,
    - a number of metrics to calculate: running time, memory usage, stack depth, and instance counts
    - two formatters that can transform framework results into graphs or templated documentation (like HTML or Latex), and
    - a flexible implementation that allows designers to extend the framework as needed with new metrics and formatters.

2. In Chapter 5, I showed through a number of experiments that the framework introduces reasonably little overhead and gives predictable results. The results found for both the bloopers and best practices matched the expectations of performance experts, and were consistent with our intuition.

3. The performance metric results obtained from prototypes and the production implementatio of the I-Help discussion system described in Chapter 6 were not the same, but were consistent estimations. The framework also allows designers to explicitly isolate the performance of portions of an application by writing application-specific design test suites and tuning framework parameters to reduce the overhead of collecting performance metrics.

4. In writing design test suites, while the framework is important for some situations, I found that the process provides guidance and delivers value even when the DTPT framework was not used. Some test cases are easier to express outside of the framework using ad-hoc tests; an example can be found in Chapter 6. These test suites still formalize performance concerns as a collection of test cases and are consistent with the DTPT process, but by avoiding the structure introduced by the framework they can be faster to write for some situations. Even when not using the framework, designers can gain real benefits by thinking about performance tests in terms of the introduced process as it explicitly considers instrumentation overhead, the execution environment, test case parameters, and alternative implementations.

8.3 Future Work

This work could be extended to provide even more benefit to the software development process. In this section, I consider some of the ways this work could be extended in the future including:

1. Perform a field study to see whether practitioners get real benefits from using the process or framework.

2. Through experimenting with large systems, gather a better understanding of the how prototype results scale to represent large implementations.

3. Add new metrics to the framework, allowing designers and developers to better track performance trade offs.

4. Explore distributed test suite execution for testing performance variability across platforms and/or to reduce the total running time of large test suites.

8.3.1 Field Study

In my thesis, I applied the framework to hypothetical design decisions in the I-Help Discussion System and reviewed the results with one of the key developers on that project. To fully understand how the consequences of using the framework in a real situation, one would need to find a real software project, identify some of the key design decisions, and let the designers and developers of that application use the framework to explore the performance trade offs of those design decisions.

This field study would provide valuable information on how well the framework integrates into practitioners' existing tool chains as well as feedback on the framework implementation and the process of design-time performance testing.

8.3.2 Scaling

When testing complex systems, there is a risk that test results obtained for a small, mock up situation or with only a sample of the true load may not scale predictably to the system's actual runtime performance. In my work, executing prototypes in design test suites to compare alternatives, it would be important to understand to what extend we can expect these results to scale, and how to identify situations in which scaling will not be as expected.

Westland developed mathematical relationships to explain how results obtained from prototypes of database driven information systems can be scaled up to predict full-scale implementation results [51]. It may be possible to apply the same lessons learned by Westland (for general-case information systems) to specific results obtained using the process or framework. If this is the case, then we may be able to provide a set of guidelines for how to interpret process and framework results as we currently leave this interpretation entirely up to the designer.

8.3.3 Other Metrics

The framework implements a number of common performance metrics including: total running time, thread execution time, stack depth, and heap memory usage. While these general metrics can give us some performance insight on a wide variety of situations, understanding the performance consequences of some design decisions may be far easier given a custom performance metric.

Javana offers a multitude of interesting metrics related to the execution of the virtual machine including: garbage collection, object instantiation, method compilation, and code optimization [31]. The framework already allows developers to implement and gather customized metrics by implementing a metric interface, but it would be useful to include more metrics with the framework. Furthermore, it would be very interesting to explore what makes a given metric valuable in a specific situation.

8.3.4 Distributed operation

Duarte et al. considered how distributed execution of unit tests can reduce the total execution time required to run nightly unit tests for continuous integration [17]. Extending the framework to allow for automated distributed execution of design test suites would not only allow designers to reduce a test suite's total execution time, but also automatically execute the design test suites on multiple platforms.

Developing software for multiple platforms is a reality in today's software. Software designers work on a wide vareity of platforms, from hand-held devices like a BlackBerry or iPod with limited RAM, to a quad core workstation with gigabytes of RAM, or a Playstation 3 with nine execution cores. Being able to write a design test suite along with a set of prototype alternatives and automatically distribute those to diverse platforms for execution and document the results would allow designers to understand the platform differences and how they will manifest in their specific application. This way, designers can compare the cost of developing platform-specific implementations or using a general multi-platform approach with the performance trade-offs of both solutions.

8.4 Summary

In this thesis, I considered design-time performance testing: investigating how software designers can estimate the performance of design alternatives early in the development cycle to inform their design decisions. I introduced a process for comparing alternatives by collecting performance metrics while running design alternative prototypes in the context of design scenario test suites. The process gives results early in the development cycle based on the prototype implementations that are consistent with our expectations and consistent with those results obtained from a full scale implementation. Designers can use the process with the framework implementation to compare design alternatives, document the results, and ultimately inform their design decisions with performance information.

References

[1] Lada A. Adamic. Zipf, power-laws, and pareto - a ranking tutorial. Online Tutorial, 2002. http://en.wikipedia.org/ wiki/Zipf%27s_law.

[2] Amazon.com. Amazon web services. Website. http://www.amazon.com/gp/ browse.html?node=3435361.

[3] Amazon.com, Inc. Amazon elastic compute cloud. Amazon Web Services Website. http://www.amazon.com/gp/browse.html?node=201590011.

[4] Apple Inc. Developer tools: Xray. Mac OS X Leopard Information Website, 2006. http://www.apple.com/ macosx/leopard/ developer/xray.html.

[5] Robert D. Atkinson and Randolph H. Court. The new economy index. Progressive Policy Institute Technology Project, 2001. http://www.neweconomyindex.org/section1_page12.html.

[6] Kurt Bittner and Ian Spence. *Managing Iterative Software Development Projects*, page 80. Addison-Wesley, 2006.

[7] Nick Bradbury. Youtunes: An example yahoo! pipe. Personal Blog. http://nick.typepad.com/blog/2007/02/youtunes_an_exa.html.

[8] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM '99: Proceedings of the Eighteen Anual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134, New York, NY, USA, 1999. IEEE Computer Society.

[9] Lionel C. Briand. Software documentation: How much is enough? In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[10] Bryan Cantrill, Michael W. Shapiro, and Adam H Leventhal. Dynamic instrumentation of production systems. In *USENIX 2004 Annual Technical Conference: Proceedings of the General Track*, pages 15–28, 2004.

[11] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261, New York, NY, USA, 2002. ACM Press.

[12] ComponentSource. Credit card authorization components. Website. http://www.componentsource.com/ features/card-authorization/ index.html.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, USA, second edition edition, 2001.

[14] Ward Cunningham and Jim Shore. Fit: Framework for integrated test. Project Website, 2002. http://fit.c2.com/.

[15] Siddhartha R. Dalal, Michael S. Hamada, and Tzyh-Jong Wang. How to improve performance of software systems: A methodology and a case study for tuning performance. *Ann. Softw. Eng.*, 8(1-4):53–84, 1999.

[16] Giovanni Denaro, Andrea Polini, and Wolfgang Emmerich. Early performance testing of distributed software applications. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 94–103, New York, NY, USA, 2004. ACM Press.

[17] Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, and Patricia Machado. GridUnit: software testing on the grid. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 779–782, New York, NY, USA, 2006. ACM Press.

[18] Eclipse test & performance tools platform. Eclipse TPTP Project Information Website. http://www.eclipse.org/ tptp/home/project_info/ general/whatisTPTP.php.

[19] Michael Feathers. Framework for integrated test: Beauty through fragility. In Andy Oram and Greg Wilson, editors, *Beautiful Code*, pages 75–84. O'Reilly, 2007.

[20] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, New York, NY, USA, 2002. ACM Press.

[21] Google. Google Maps API. Google Code Website. http://code.google.com/apis/ maps/index.html.

[22] Google. OpenSocial API. Google Code Website. http://code.google.com/apis/opensocial/.

[23] Elliotte Rusty Harold. Correct, beautiful, fast (in that order): Lessons from designing xml verifiers. In Andy Oram and Greg Wilson, editors, *Beautiful Code*, pages 59–74. O'Reilly, 2007.

[24] ihelp discussion system. Project Website. http://ihelp.usask.ca/.

[25] ISO. International Standard ISO/IEC 9126. Information technology – Software product evaluation – Quality characteristics and guidelines for their use. *International Organization for Standardization, International Electrotechnical Commission*, 1991.

[26] Ryszard Janicki, David Lorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In *Software Fundamentals: Collected Papers by David L. Parnas*, pages 71–88. Addison-Wesley Professional, 2001/1997.

[27] Yan Jin, Antony Tang, Jun Han, and Yan Liu. Performance evaluation and prediction for legacy information systems. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 540–549, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Douglas Kramer. API documentation from source code comments: a case study of Javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA, 1999. ACM Press.

[29] Chandra Krintz. Coupling on-line and off-line profile information to improve program performance. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 69–78, Washington, DC, USA, 2003. IEEE Computer Society.

[30] Raimondas Lencevicius and Edu Metz. Performance assertions for mobile devices. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 225–232, New York, NY, USA, 2006. ACM Press.

[31] Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: a system for building customized java program analysis tools. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 153–168, New York, NY, USA, 2006. ACM Press.

[32] Minimalist gnu for windows. Project Website. http://www.mingw.org/.

[33] John O'Hanley. Java practices. Website. http://www.javapract ices.com/.

[34] Michael Olan. Unit testing: test early, test often. *J. Comput. Small Coll.*, 19(2):319–328, 2003.

[35] David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly. Profiling with AspectJ. *Software Practice & Experience*, 37(7):747–777, 2007.

[36] Performance bloopers. Eclipsepedia Wiki, 2006. http://wiki.eclipse.org/ index.php/Performance_Bloopers.

[37] Valentina Popescu. Java application profiling using TPTP. Eclipse Corner Article, 2006. http://www.eclipse.org/ articles/Article-TPTP-Profiling-Tool/ tptpProfilingArticle.html.

[38] Vijay Ramachandran. Design patterns for optimizing the performance of J2EE applications. Sun Developer Network, 2001. http://java.sun.com/ developer/technicalArticles/ J2EE/J2EEpatterns/.

[39] Jakub Rudzki and Tarja Systä. Performance implications of design pattern usage in distributed applications: case studies in J2EE and .NET. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 1–11, New York, NY, USA, 2006. ACM Press.

[40] Jayshankar Sankarasetty, Kevin Mobley, Libby Foster, Tad Hammer, and Terri Calderone. Software performance in the real world: personal lessons from the performance trauma team. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance*, pages 201–208, New York, NY, USA, 2007. ACM Press.

[41] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*, pages 84–85. John Wiley & Sons Inc., New York, NY, USA, sixth edition, 2002.

[42] Adam Slosarski. NTime - performance unit testing tool. The Code Project .NET, 2004. http://www.codeproject.com/ dotnet/ntime.asp.

[43] David Gerard Sullivan. *Using Probabilistic Reasoning to Automate Software Tuning.* PhD thesis, Harvard University, 2003. http://www.eecs.harvard.edu/s̃ullivan/thesis.pdf.

[44] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108, New York, NY, USA, 2001. ACM Press.

[45] Sun Microsystems Inc. Sun grid compute utility. Sun Services Website. http://www.sun.com/service/sungrid/index.jsp.

[46] Sun Microsystems Inc. JVM$^{\text{TM}}$ Tool Interface. J2SE 1.5.0 Documentation, 2004. http://java.sun.com/ j2se/1.5.0/docs/guide/ jvmti/jvmti.html.

[47] Haonan Tan, Derek L. Eager, Mary K. Vernon, and Hongfei Guo. Quality of service evaluations of multicast streaming protocols. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and modeling of computer systems*, pages 183–194, New York, NY, USA, 2002. ACM Press. http://citeseer.ist.psu.edu/tan02quality.html.

[48] TPC. Transaction processsing performance council. Organization Website. http://www.tpc.org.

[49] Alexander Ufimtsev and Liam Murphy. Performance modeling of a javaee component application using layered queuing networks: revised approach and a case study. In *SAVCBS '06: Proceedings of the 2006 conference on Specification and verification of component-based systems*, pages 11–18, New York, NY, USA, 2006. ACM Press.

[50] Dimitri van Heesch. Doxygen. Project website, 1997. http://www.stack.nl/~dimitri/doxygen/.

[51] J. C. Westland. Scaling up output capacity and performance results from information systems prototypes. *ACM Trans. Database Syst.*, 15(3):341–358, 1990.

[52] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society.

[53] Yahoo! Inc. Pipes: Rewire the web. Yahoo! Website, 2007. http://pipes.yahoo.com/pipes/.

[54] Andrew Zhang. p-unit - an open source framework for performance benchmark and unit test. Sourceforge Project, 2007. http://p-unit.sourceforge.net/.

[55] Min Zhao, Bruce R. Childers, and Mary Lou Soffa. A model-based framework: An approach for profit-driven optimization. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 317–327, Washington, DC, USA, 2005. IEEE Computer Society.

Appendix A

Example Source Code Listings

This appendix includes source code listings written in Java and are referenced by examples and pseudocode throughout the thesis.

A.1 Sorting Test Suite Driver

```
public class SortingDriver {
 public static void main(String[] args) {
  // instantiate the test suite
  TestSuite s = suiteMain();
  // execute the test suite
  s.execute();
  // pass the results to a graph formatter
  FormattedResults fr = new GraphFormatter();
  fr.results(s.results());
  fr.xaxis(1); // the run param on the x-axis
  fr.series(3); // the run param that identifies a series
  fr.yaxis(0); // the metric value on the y-axis
  fr.format("comparison.png");
 }
 public static TestSuite suiteMain() {
  Object[][] runParams = new Object[4][];
  TestRun tr;
  TestSuite ts = new TestSuite("edu.usask.cs.hopkins.sorting.testing");
  // setup the test run
  tr = new TestRun(new TestSorter());
  ts.addRun(tr);
  // add the metrics we wish to capture
  tr.addMetric(new TotalRunningTime());
  // set the testrun timeout in milliseconds
  tr.timeout(4000);
  // no case parameters
  tr.caseParams(new Object[0]);
  tr.iterations(2);
  // setup the run parameters
  runParams[0] = new Integer[] {10};
  runParams[1] = new Integer[] {10, 100, 1000, 2000, 5000, 10000, 20000};
  runParams[2] = new Integer[] {10000};
  runParams[3] = new Sorter[] {new BubbleSorter(), new SelectionSorter(),
    new InsertionSorter(), new HeapSorter(),
    new MergeSorter(), new QuickSorter(),
    new RadixSorter(), new StandardSorter()};
  tr.runParams(runParams);
  return ts;
 }
}
```

A.2 Sorting Test Case

```
public class TestSorter extends TestCase {
 // Random number generator
 private static Random rand = new Random();
 // Override the run method, call doTest with typed parameters
 public void run (Object[] params) {
  doTest(Utils.objectToInt(params[0]), Utils.objectToInt(params[1]),
    Utils.objectToInt(params[2]), (Sorter) params[3]);
 }
 // execute the test case given given a set of parameters
 private void doTest(int count, int length, int max, Sorter s) {
  Comparable[] list = new Comparable[count];
  for (int i = 0; i < count; i++) {
   fillList(list, max);
   s.sort(list);
  }
 }
 // fill a list with random integers less than max
 private void fillList(Comparable[] list, int max) {
  for (int i = 0; i < list.length; i++) {
   list[i] = new Integer(rand.nextInt(max));
  }
 }
}
```

A.3 Quick Sort Implementation

```
public class QuickSorter implements Sorter {
 private Random r;
 public QuickSorter() {
  r = new Random();
 }
 public void sort(Comparable[] list) {
  sortR(list, 0, list.length);
 }
 // recursive sort implementation
 public void sortR(Comparable[] list, int min, int max) {
  int i, pivotPos;
  Comparable tmp, pivot;

  pivotPos = selectPivot(list, min, max); // find a pivot
  if ((pivotPos < min) || (pivotPos >= max)) {
   throw new RuntimeException("Bad Pivot");
  }

  pivot = list[pivotPos]; // place pivot at end of list
  list[pivotPos] = list[max - 1];

  // move items < pivot to the front of the list
  pivotPos = min;
  for (i = min; i < max - 1; i++) {
   if (list[i].compareTo(pivot) <= 0) {
    tmp = list[i];
    list[i] = list[pivotPos];
    list[pivotPos] = tmp;
    pivotPos++;
   }
  }

  // put the pivot back between both lists
  list[max-1] = list[pivotPos];
  list[pivotPos] = pivot;

  // sort each sub list recursively
  if (pivotPos - min > 1) {
   sortR(list, min, pivotPos);
  }
  if (max - pivotPos > 1) {
   sortR(list, pivotPos, max);
  }
 }

 public int selectPivot(Comparable[] list, int min, int max) {
  return r.nextInt(max-min) + min; // select a random pivot position
 }
}
```

A.4 Storage Component Test Case

```
public class TestComponent extends TestCase {
 // Random number generator
 private static Random rand = new Random();

 // Override the run method, call doTest with typed parameters
 public void run (Object[] params) {
  doTest(Utils.objectToInt(params[0]), Utils.objectToInt(params[1]),
    Utils.objectToInt(params[2]), (StorageManager) params[3]);
 }
 // execute the test case given given a set of parameters
 private void doTest(int threads, int requestsPerThread,
    int posts, StorageManager datasource) {
  Thread[] threads = new Thread[threads];
  int i;

  // connect to the datasource
  datasource.connect();

  // initialize the datasource
  datasource.initialize();

  // generate a data set
  DataGenerator.generate(datasource, posts);

  // start up each thread to make requests of the data source
  for (i = 0; i < threads; i++) {
   threads[i] = new Thread(new RequestMaker(requestsPerThread, datasource));
   threads[i].start();
  }

  // wait for each thread to finish making requests
  for (i = 0; i < threads; i++) {
   threads[i].join();
  }

  // disconnect form the datasource
  datasource.disconnect();
 }
}
```

A.5 Cache Test Suite Driver

```
public class CacheDriver {
 public static void main(String[] args) {
  // instantiate the test suite
  TestSuite s = suiteMain();
  // execute the test suite
  s.execute();
  // pass the results to a table formatter
  // not shown here
 }
 public static TestSuite suiteMain() {
  Object[][] runParams = new Object[4][];
  TestRun tr;
  TestSuite ts = new TestSuite("edu.usask.cs.hopkins.ihelp.cache.testing");
  // setup the test run
  tr = new TestRun(new TestComponent());
  ts.addRun(tr);
  // add the metrics we wish to capture
  tr.addMetric(new TotalRunningTime());
  // set the testrun timeout in milliseconds
  tr.timeout(4000);
  // no case parameters
  tr.caseParams(new Object[0]);
  tr.iterations(2);
  // setup the run parameters
  runParams[0] = new Integer[] {10, 100};
  runParams[1] = new Integer[] {10, 100};
  runParams[2] = new Integer[] {100, 1000};
  runParams[3] = new Sorter[] {new MySQLManager(),
   new CacheManager(new MySQLManager(), 10),
   new CacheManager(new MySQLManager(), 50)};
  tr.runParams(runParams);
  return ts;
 }
}
```

A.6 Caching Test Suite Driver

```
public class CacheDriver {
 public static void main(String[] args) {
  // instantiate the test suite
  TestSuite s = suiteMain();
  // execute the test suite
  s.execute();
  // pass the results to a table formatter
  // not shown here
 }
 public static TestSuite suiteMain() {
  Object[][] runParams = new Object[4][];
  TestRun tr;
  TestSuite ts = new TestSuite("edu.usask.cs.hopkins.ihelp.cache.testing");
  // setup the test run
  tr = new TestRun(new TestComponent());
  ts.addRun(tr);
  // add the metrics we wish to capture
  tr.addMetric(new TotalRunningTime());
  // set the testrun timeout in milliseconds
  tr.timeout(4000);
  // no case parameters
  tr.caseParams(new Object[0]);
  tr.iterations(2);
  // setup the run parameters
  runParams[0] = new Integer[] {10, 100};
  runParams[1] = new Integer[] {10, 100};
  runParams[2] = new Integer[] {100, 1000};
  runParams[3] = new Sorter[] {new MySQLManager(),
   new CacheManager(new MySQLManager(), 10),
   new CacheManager(new MySQLManager(), 50)};
  tr.runParams(runParams);
  return ts;
 }
}
```