Environmental Modelling & Software 26 (2011) 1240–1250

Contents lists available at ScienceDirect

Environmental Modelling & Software

journal homepage: www.elsevier.com/locate/envsoft

Environmental modeling framework invasiveness: Analysis and implications

W. Lloyd [a,b,*], O. David [a,b], J.C. Ascough II [c], K.W. Rojas [d], J.R. Carlson [d], G.H. Leavesley [a], P. Krause [e], T.R. Green [c], L.R. Ahuja [c]

[a] Dept. of Civil and Environmental Engineering, Colorado State University, Fort Collins, CO 80523, USA
[b] Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523, USA
[c] USDA-ARS, ASRU, 2150 Centre Ave., Bldg. D, Suite 200, Fort Collins, CO 80526, USA
[d] USDA-NRCS, 2150 Centre Ave., Bldg. A, Fort Collins, CO 80526, USA
[e] Department of Geography, Friedrich-Schiller-Universität Jena, Jena, Germany

ARTICLE INFO

Article history:
Received 4 May 2010
Received in revised form
24 February 2011
Accepted 28 March 2011
Available online 12 May 2011

Keywords:
Component-based modeling
Environmental modeling frameworks
Invasiveness
Frameworks
Software metrics

ABSTRACT

Environmental modeling frameworks support scientific model development by providing model developers with domain specific software libraries which are used to aid model implementation. This paper presents an investigation on the framework invasiveness of environmental modeling frameworks. Invasiveness, similar to object-oriented coupling, is defined as the quantity of dependencies between model code and a modeling framework. We investigated relationships between invasiveness and the quality of modeling code, and also the utility of using a lightweight framework design approach in an environmental modeling framework. Five metrics to measure framework invasiveness were proposed and applied to measure dependencies between model and framework code of several implementations of Thornthwaite and the Precipitation-Runoff Modeling System (PRMS), two well-known hydrological models. Framework invasiveness measures were compared with existing common software metrics including size (lines of code), cyclomatic complexity, and object-oriented coupling. Models with lower framework invasiveness tended to be smaller, less complex, and have less coupling. In addition, the lightweight framework implementations of the Thornthwaite and PRMS models were less invasive than the traditional framework model implementations. Our results show that model implementations with higher degrees of framework invasiveness also had structural characteristics which previously have been shown to predict poor maintainability, a non-functional code quality attribute of concern. We conclude that using a framework with a lightweight framework design shows promise in helping to improve the quality of model code and that the lightweight framework design approach merits further attention by environmental modeling framework developers.

1. Introduction

For an environmental model, the software life-cycle begins when a model developer initially writes scientific modeling code to represent a particular natural system or phenomena. The software life-cycle ends when the modeling code is retired and replaced by a new modeling paradigm or approach. Throughout the software life-cycle of a scientific model, source code may need to be modified to reflect improvements in understanding made by the scientific community. Additionally, the languages, libraries, and modeling frameworks used by research organizations may periodically change and the modeling code must be adaptable to run under new environments. Furthermore, researchers collaborate within and across organizations in an effort to improve and share scientific knowledge. Shared model code must be reusable by organizations which may not have access to the same computing environments, libraries, frameworks, languages, and software development expertise. Because of the challenges encountered in developing and supporting scientific models throughout their software life-cycle, numerous software frameworks have been developed.

A software framework is a software library which provides domain specific functionality in a reusable form (Gamma et al., 1995). Software frameworks help define the software architecture of applications by: 1) providing a reusable design which guides software developers in partitioning functionality into units, commonly referred to as components, classes, or modules; and 2) specifying how units communicate and manage the thread of execution. Frameworks enable design reuse and can be classified as

* Corresponding author. Present address: USDA-ARS, ASRU, 2150 Centre Ave., Bldg. D, Suite 200, Fort Collins, CO 80526 USA. Tel.: +1 970 492 7311; fax: +1 970 492 7310.
E-mail address: wlloyd@acm.org (W. Lloyd).

1364-8152/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.envsoft.2011.03.011

either non-domain specific or domain specific. Non-domain specific frameworks provide support for general software architecture elements such as database access, enterprise services, graphical interface design, and transaction management. Domain specific frameworks provide reusable design and functionality for specific problem domains such as industrial control systems, networking/telecommunications, inventory tracking, and environmental modeling frameworks, which are the focus of this study.

Environmental modeling frameworks support model development by providing libraries of core environmental modeling modules or components which support: component interaction and communication, spatial-temporal stepping and iteration, up/down-scaling of spatial data, multi-threading/multiprocessor support, and cross language interoperability, as well as reusable tools for data analysis and visualization. Environmental modeling frameworks also provide structure for models by supporting the disaggregation of modeling functions into components, classes, or modules. In this paper, we refer to functional units of model code as components. Components, once implemented in a particular framework, are able to be reused in other models coded to the same framework with little migration effort. One advantage of using a common environmental modeling framework is that pre-existing modules or components may exist in a library which can help facilitate model development (Voinov et al., 2004; Argent et al., 2006).

A variety of environmental modeling frameworks exist and have been under development using both procedural and object-oriented programming languages for more than a decade, including early frameworks such as the Interactive Component Modelling System (ICMS) (Rizzoli et al., 1998; Reed et al., 1999; Argent, 2005), Tarsier (Watson and Rahman, 2004), the Spatial Modelling Environment (SME) (Maxwell, 1999; Voinov et al., 1999), the Modular Modeling system (MMS) (Leavesley et al., 2002, 2006) and TIME (Rahman et al., 2003, 2004). The Earth System Modeling Framework (ESMF, Collins et al., 2005) is an example of an environmental modeling framework implemented using procedural languages. Object-oriented programming became popular in the late 1990s and early 2000s which led to innovations in framework design. Newer environmental modeling frameworks have typically been implemented in object-oriented languages such as C++, C#, or Java and take advantage of object-oriented programming principles. These frameworks began to be known as component-based frameworks by offering libraries of "pluggable" modeling components. With component-based frameworks, various aspects of model implementation can be quickly changed, leading to easier model refinement and expansion. Among environmental modeling frameworks, there does not appear to be a clear distinction between object-oriented frameworks and component-based frameworks. Nearly all of the environmental modeling frameworks developed with object-oriented technologies appear to support component-based software development by offering the capability of having pluggable components or classes which can readily be substituted. This research investigation primarily utilizes component-based frameworks including the Object Modeling System (OMS, 2010) 2.2 and 3.0 (David et al., 2002, 2010), the Common Component Architecture (CCA) 0.6.6 (Armstrong et al., 1999), and the Open Modeling Interface (OpenMI) 1.4 (Blind and Gregersen, 2005). Jagers (2010) presented a functional comparison of modeling frameworks with the objective of qualitatively identifying framework similarities and differences.

Donatelli and Rizzoli (2008) highlighted modeling component dependence on modeling frameworks. They suggested that modeling components should be developed with a generic interface (i.e., not framework specific) to enhance reuse opportunities and make unit testing easier to accomplish. Practical experience in using environmental modeling frameworks has shown that model applications heavily dependent on a framework are hard to reuse, maintain, and repurpose outside of the framework context. In this paper, we define the degree of dependency between an environmental modeling framework and model code as "framework invasiveness." This is the degree to which model code is coupled to the underlying framework. Framework to modeling code invasiveness occurs due to the following:

- Use of a framework Application Programming Interface (API) consisting of data types and methods/functions with which developers interface to harness framework functionality;
- Use of framework specific data structures (e.g., classes, types, constants);
- Implementation of framework interfaces and extension of framework classes;
- Boilerplate code (e.g., "non-science" code that is required so that a model can run under a specific environmental modeling framework);
- Framework requirements including language, platform, and libraries; and
- Organizational investment (e.g., training, financial, development).

One goal of this research is to explore relationships between environmental model code quality and the degree of invasiveness between model code and environmental modeling frameworks. Framework to application invasiveness is a type of code coupling. Object-oriented coupling (i.e., coupling between classes in an object-oriented program) has been shown to correlate inversely with software fault proneness where fault proneness is the likelihood of a mistake in the code (Basil et al., 1996; Briand et al., 1999, 2000). Mistakes in model code negatively impact the functional correctness of the code, thereby reducing the functional aspects of code quality. There are other important dimensions to model code quality, such as maintainability and portability, which are referred to as non-functional quality attributes. For this study, we are primarily interested in understanding the impact of framework invasiveness on non-functional quality attributes of the model code. We are not assessing how frameworks specifically impact the functional accuracy of models as this largely depends on the developer implementing scientific algorithms properly. Previous research has shown that both code size and coupling can be used to predict code maintainability (Li and Henry, 1993; Dagpinar and Jahnke, 2003; Anda, 2007). This study focuses on quantifying the code invasiveness incurred by using environmental modeling frameworks because we believe this may be suggestive of the non-functional quality of model code. The impact of invasiveness can be considered as the "overhead" imposed by using a particular framework for a specific modeling problem. For the remainder of this paper, when we refer to model code quality we are referring to non-functional quality attributes and not functional correctness of the model. Ultimately, we are interested in understanding how framework invasiveness impacts non-functional code quality attributes such as:

- Maintainability - the ease of maintaining program code for bug fixes, feature enhancements, and upgrading to new framework versions (Dig and Johnson, 2006). Framework dependencies may complicate bug fixes and feature enhancements by increasing the effort required to understand and make application changes. Framework version upgrades can be complex if substantial changes occur to the framework which prevent backward compatibility. In some cases, API changes between framework versions creates a substantial barrier for upgrading existing code bases.
- Portability/Reusability - the ease of porting application code for use outside the framework or for use in other frameworks.

Dependencies from using framework APIs clutter code with framework specific constructs which must be adapted in order to reuse code outside the original framework. Porting code may involve adapting to another framework, or to run standalone outside a framework. Porting may also involve adapting code from one language, such as FORTRAN, to another language, such as Java.

- Understandability - the ability for developers new to the code to understand the implementation. Code with a high degree of framework dependencies may be cluttered, potentially making the modeling logic harder to identify and comprehend. Understandability is a factor which impacts maintainability, portability, and reusability.

A second goal of this research is to explore the utility of a lightweight framework design approach for use in scientific and environmental modeling frameworks. Modeling frameworks can be classified as traditional or lightweight based on various design characteristics (Richardson, 2006a). General characteristics of these types of frameworks are described in Table 1. The primary difference between traditional and lightweight frameworks is how they present functionality to the developer. Traditional frameworks, also known as object-oriented frameworks (e.g., Java's Swing Application Framework for GUI development), provide developers with an API that is often large, and developers typically spend considerable time becoming familiar with framework APIs before writing model code. Lightweight frameworks aim to reduce dependencies between business/model code and framework code by offering alternative ways to harness framework functionality other than through the use of a large programming API.

The lightweight framework design approach is a new approach to framework design that originated from various web application and enterprise frameworks (Richardson, 2006b). Our research aims to investigate the utility of harnessing this approach for scientific and environmental modeling frameworks. A variety of techniques are used including programming annotations that capture metadata to identify specific points in the model code where framework functionality should be integrated, and also through the use of external XML files. Wherever possible, "convention over configuration" is favored such that system defaults are assumed and developers are only required to specify unconventional details in model code. Non-default behavior may include, for example, unique component data input/output requirements, pre-conditions and post-conditions. Framework-specific data types that override system types are avoided in lightweight framework designs.

The use of inversion of control principles (Fowler, 2004) is the fundamental defining difference between traditional frameworks and lightweight frameworks. In general, inversion of control is the idea that the model code should not directly invoke framework APIs, but control is reversed and the framework injects functionality into the model code where it is needed. The model developer specifies where to inject functionality at specific points in the code by using language annotations. Alternatively the locations in model code where framework functionality is injected may be assumed based on some predefined default or on analysis of the code structure itself, a concept derived from aspect-oriented programming (Elrad et al., 2001).

The broad objectives of this research are to measure framework invasiveness in order to explore the implications on non-functional attributes of model code quality, and to explore the utility of the lightweight framework approach for scientific and environmental modeling. Specifically, we seek to investigate the following research questions: 1) how do we quantify and measure framework invasiveness and what is the impact of this invasiveness on the non-functional aspects of model code quality, and 2) what is the utility of using lightweight environmental modeling frameworks for model development? A better understanding of the phenomenon of framework to application invasiveness may help modelers in choosing and designing modeling frameworks to improve the quality of scientific models throughout their entire software lifecycles.

2. Methods and materials

2.1. Environmental modeling frameworks

We performed a case study using two environmental models, a monthly water balance model (based on Thornthwaite, 1948) and a complex watershed-scale model (the Precipitation Runoff Modeling System, PRMS, Leavesley et al., 1983). Thornthwaite was an ideal candidate for the study because it features a typical structure for a hydrological simulation model and its size and complexity were manageable for porting to a variety of frameworks. PRMS augmented the study with a larger-scale scientific model which is in wide use. We used five environmental modeling frameworks actively under development: the Earth System Modeling Framework (ESMF) 3.1.1 (FORTRAN and C versions), the Common Component Architecture (CCA) 0.6.6, the Open Modeling Interface (OpenMI) 1.4, and the Object Modeling System (OMS) versions 2.2 and 3.0. It should be noted that the CCA is a more general scientific modeling framework which has been designed to be broadly applicable, whereas the other frameworks studied here were primarily proposed and developed within the environmental modeling research community. Several common structural measures were used to assess attributes of the environmental model implementations including size, complexity, and object-oriented coupling. Furthermore, a new set of software metrics was devised and applied to quantify the invasiveness between the framework and model code. The frameworks listed above were used to implement Thornthwaite. The OMS 2.2 and 3.0 frameworks were used to implement PRMS. The programming languages and frameworks used for the Thornthwaite and PRMS implementations are summarized in Table 2. Additionally, three non-framework based implementations of Thornthwaite were implemented using Java, C++, and FORTRAN (Table 2) to provide a starting point for developing the framework-based versions.

ESMF is an open source framework developed by the National Center for Atmospheric Research (NCAR) for building climate, numerical weather prediction, data assimilation, and other Earth science software applications (Collins et al., 2005). ESMF is procedural in nature and supports model development using the FORTRAN and C programming languages. CCA was developed by the members of the Common Component Architecture Form, and is a component architecture for high performance computing. Features of the CCA include multi-language, multi-dimensional arrays, and a variety of network data transports not typically suited for wide area networks (Armstrong et al., 1999). OpenMI was originally developed by a European research group known as the 5th EU Framework Programme (1998–2002) and is a software component interface definition for developing models in the water

Table 1
Traditional versus lightweight framework design classification.

Traditional Frameworks	Lightweight Frameworks
Components under the framework are:	Components under the framework are:
▪ bound statically at compile time ▪ tightly coupled to the framework by extension of framework classes, implementation of framework interfaces, use of framework specific data types/classes, and use of framework specific functions/methods • Framework provides specialized versions of native language data types • Framework has a "large" programming interface (API) • Framework use may depend on many libraries	▪ bound dynamically at run time by use of language annotations/dependency injection techniques (inversion of control software design pattern) ▪ loosely coupled and largely independent of the framework • Convention over configuration: developers only specify unconventional details in code as defaults are otherwise assumed • Framework uses native language data types • Framework has a "small" programming interface (API)

Table 2
Languages and frameworks for Thornthwaite and PRMS model implementation.

Model	Language	Framework	Base Implementation
Thornthwaite	Java	CCA 0.6.6	OMS 3.0 Java
	C	ESMF 3.1.1	C++, no framework
	Fortran	ESMF 3.1.1	FORTRAN, no framework
	C++	None	OMS 3.0 Java
	FORTRAN	None	OMS 3.0 Java
	Java	None	OMS 3.0 Java
	Java	OMS 2.2	developed before experiment
	Java	OMS 3.0	developed before experiment
	Java	OpenMI 1.4	OMS 3.0 Java
PRMS	Java	OMS 2.2	developed before experiment
	Java	OMS 3.0	developed before experiment

domain (Blind and Gregersen, 2005). OpenMI supports rescaling temporal/spatial data so that components operating on data with different geometries can interoperate seamlessly. The OpenMI Thornthwaite model in this study was developed using a Java-based implementation of OpenMI, although a .NET/C# version exists and is generally considered more popular than the Java-based implementation. The Object Modeling System (OMS) versions 2.2 and 3.0 were developed by the USDA – Agricultural Research Service (ARS) in cooperation with Colorado State University. OMS facilitates component-oriented simulation model development in Java, C/C++ and FORTRAN (David et al., 2002, 2010), and version 2.2 provides an integrated development environment (IDE) with numerous tools supporting data retrieval, GIS, graphical visualization, statistical analysis and model calibration (Ahuja et al., 2005).

The ESMF 3.1.1, CCA 0.6.6, OpenMI 1.4, and OMS 2.2 frameworks provide an API as well as specific data types which modelers must use in order to interface with the framework. Alternatively, OMS 3.0 has been developed using a lightweight framework design approach for model development. Lightweight frameworks exist in other computing domains including web application frameworks such as Tapestry (Apache Tapestry, 2010), Spring (SpringSource, 2010), and Terracotta (Terracotta, 2010) which is a distributed parallel computation lightweight framework. In a lightweight framework design approach, modeling components are decoupled from the framework API wherever possible so that they exist as plain classes implementing only model specific logic. Furthermore, the boilerplate code (often referred to as plumbing) is typically removed by refactoring using language annotations. The lightweight framework design promises to advance environmental modeling by streamlining model source code, which can lead to better reusability and portability of framework-based model components (David et al., 2010).

2.2. Environmental models

2.2.1. Thornthwaite model

The Thornthwaite monthly water balance model simulates water allocation among components of a hydrological system (Thornthwaite, 1948). The Thornthwaite model was originally developed by the US Geological Survey (USGS) as a small-scale proof of concept prototype for a water balance component (McCabe and Markstrom, 2007). The model consists of a climate component, science components (e.g., length of daylight, evapotranspiration, snow/rain accounting, soil water balance, and runoff), a component for simulation output, and a component for model execution control (Fig. 1). The disaggregation of the Thornthwaite model into components was derived from the original non-framework based FORTRAN implementation of the model when the OMS 2.2 and OMS 3.0 Java implementations were made. This component disaggregation was carried over and replicated in each subsequent model implementation (the base model implementations are listed in Table 2). Both the OMS 2.2 and OMS 3.0 Thornthwaite Java implementations were available prior to this study with the OMS 3.0 implementation used as a base to provide the FORTRAN, C++, CCA (Java), Java, and OpenMI (Java) implementations. For the OpenMI Thornthwaite implementation, model flow control had to be substantially reworked because OpenMI 1.4 requires a "functional programming" approach to flow control versus the traditional "procedural programming" approach used by the other frameworks. Procedural programming uses a series iterative statements to change the state of the program to define the computation, whereas functional programming avoids state and mutable data by defining the computation as a series of function calls in a recursive manner (Mitchell, 1996). The plain FORTRAN model implementation was used to implement the ESMF FORTRAN model and the plain C++ model implementation was used to implement the ESMF C model. At the time of implementation, ESMF did not support native C++ so the C++ was easily converted to C as the model was simultaneously ported to the ESMF C framework.

The Thornthwaite model uses a simple monthly time step which was driven by the data input file. No special spatial aggregation/disaggregation or unit transformations were required for this model. The model was selected since it has a typical structure for a hydrological simulation model and its size and complexity were manageable for porting to a variety of available frameworks. Different versions (i.e., different programming languages) of the Thornthwaite model were implemented under the environmental modeling frameworks (Table 2); however, all model implementations were coded to produce identical numerical output. Programming language specific formatting functions were not used. The non-framework FORTRAN implementation of Thornthwaite represents the smallest complete implementation at 244 lines of code which included eight distinct modeling components. The Thornthwaite model implementation for the environmental modeling frameworks made use of only basic fundamental framework aspects including framework support for component aggregation and component interaction/communication. Monthly time stepping was driven by the data input file.

2.2.2. Precipitation-runoff modeling system (PRMS)

The precipitation-runoff modeling system (PRMS) (Fig. 2) is a deterministic, distributed-parameter model developed to evaluate the impact of various combinations of precipitation, climate, and land use on stream flow, sediment yields, and general basin hydrology (Leavesley et al., 1983). PRMS source code used in this study was based on the source code originally implemented using the Modular Modeling System (MMS) (Leavesley et al., 2002, 2006). Daily basin response to normal and extreme rainfall and snowmelt can be simulated to evaluate changes in water balance relationships, flow regimes, flood peaks and volumes, soil–water

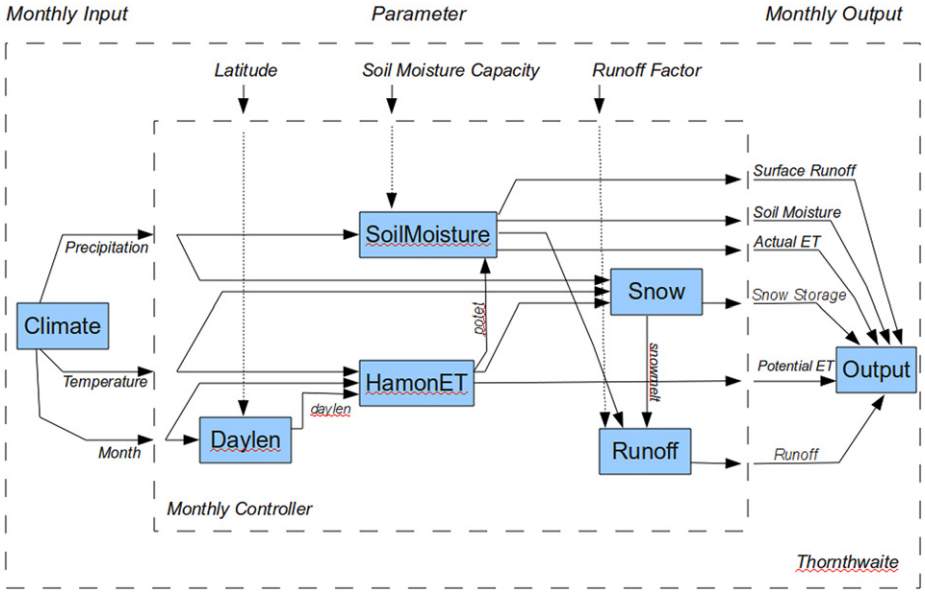

Fig. 1. Schematic of Thornthwaite model components (from McCabe and Markstrom, 2007).

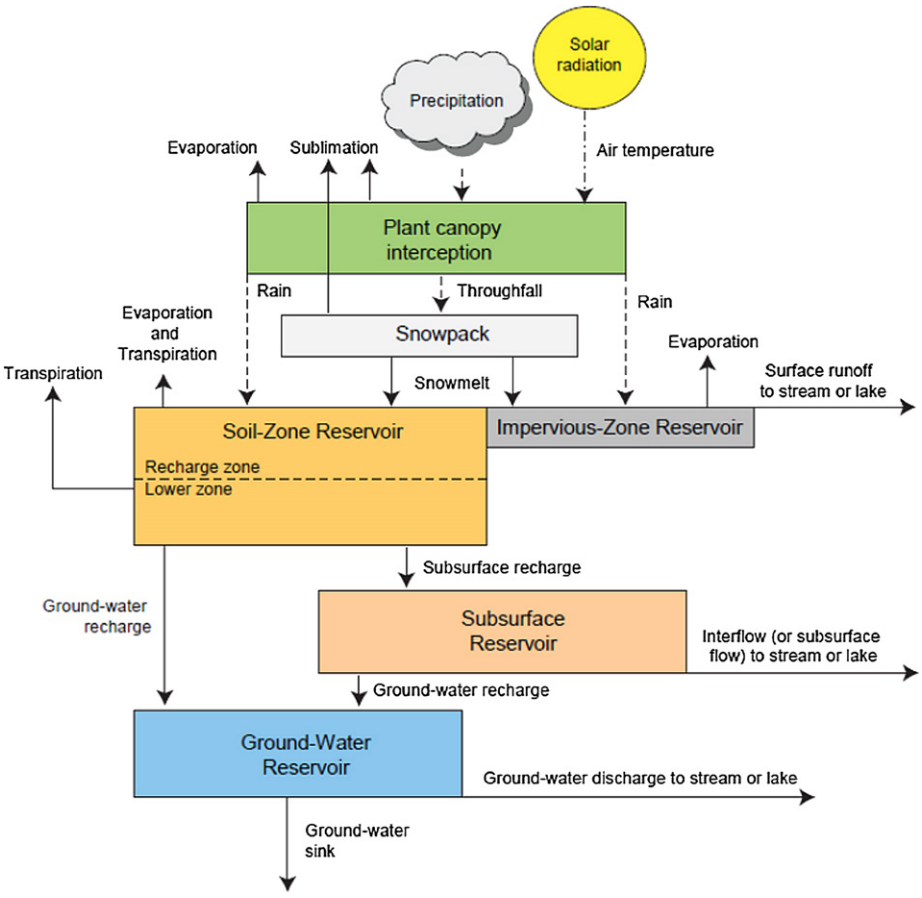

Fig. 2. Schematic of the Precipitation Runoff Modeling System (from Markstrom et al., 2008).

relationships, sediment yields, and groundwater recharge. PRMS in contrast to Thornthwaite augmented the study with a larger-scale scientific model which is in wide use. Due to the large size of the PRMS model (~17,000 lines of code), and limited time and resources, it was not feasible to port the model outside the OMS framework for this study. Only Java-based implementations in the OMS 2.2 and 3.0 frameworks were studied. Similar to Thornthwaite model framework implementations, the PRMS model implementations were coded to produce identical numerical output. The OMS 2.2 PRMS implementation consisted of 27 modeling components and approximately 17,000 lines of code. The PRMS model was disaggregated into modeling components based on the physical processes of the system with initial model disaggregation performed in the 1980s. The ability to support different approaches for modeling various physical processes such as evapotranspiration was the primary motivation for PRMS disaggregation. Both PRMS implementations (OMS 2.2 and 3.0) utilized framework support for component aggregation, interaction, and communication as well as model time stepping.

2.3. Framework invasiveness measures

Research in object-oriented software evaluation has produced numerous metrics which help to measure attributes such as the coupling, cohesion, and inheritance among classes in an object-oriented program (Chidamber and Kemerer, 1994). Several coupling measures already exist which attempt to quantify dependencies among classes in programs including coupling between object classes (CBO), efferent coupling (fan-out), afferent coupling (fan-in), response for a class (RFC), and message passing coupling (MPC). But are these coupling measures useful to quantify the dependencies between framework and modeling code? CBO and RFC have been shown to correlate positively with the quantity of software failures in object-oriented programs (Basil et al., 1996; Briand et al., 2000). Briand et al. (2000) further recommended that measuring coupling to library classes be done separately from measuring coupling to application classes as some differences are seen. In this study, we are interested in focusing on measuring the dependencies between framework classes (modules) and application (model) code. This is very similar to Briand's measurement of coupling to library classes in an object-oriented system. New metrics may help to quantify specifically the dependencies between application code and framework APIs. The following measures are proposed to quantify the invasiveness between environmental modeling frameworks and model code:

1. Framework data types used (FDT-used).
2. Framework data type uses (FDT-uses).
3. Framework functions used (FF-used).
4. Framework function type uses (FF-uses).
5. Framework dependent lines of code (FDLOC).

2.3.1. Framework data types and framework functions

There are two primary framework constructs for which we quantify usage in a model: framework data types and framework functions. We count the total number of framework data types used (FDT-used), and the total number of framework data type uses in the modeling code (FDT-uses). FDT-used counts the total number of unique framework types (classes, data structures, types, etc.) which are used in the model. FDT-uses counts the total number of uses of framework specific data types in the model. The total number of framework functions used (FF-used), and the total number of framework function uses (calls) appearing in the modeling code (FF-uses) are also counted. FF-used counts the total number of unique framework functions (functions, methods, subroutines, etc.) which are called in the model. FF-uses counts the total number of calls to framework functions in the model.

Using both framework data type constructs and framework function constructs in modeling code introduces framework dependencies. In order to reuse model code outside the framework, framework constructs must be replaced with equivalent non-framework versions. Three variations of the framework metrics can be calculated: a simple raw count, a count of framework construct usage weighted per 1000 lines of code (KLOC) (e.g., FDT/FF-used/-uses per KLOC), and the percentage of usage relative to all framework constructs used/uses in the application code (e.g., % FDT/FF-used/-uses). The raw count can be used to compare the number of framework constructs used/uses among different framework-based model implementations. Normalizing by KLOC allows the density of framework construct usage between models to be compared using a common code size. The percentage of framework constructs (FDT or FF) used/uses versus all constructs (FDT or FF) used/uses in a program can help us understand the ratio of usage that depends on the framework. This latter metric is rather tedious to calculate since it requires a complete analysis of constructs (data types or functions) for the entire application. For this study when counting total constructs used/uses, constructs which were declared by the modeling code were not included in the total count of constructs used.

2.3.2. Framework dependent lines of code (FDLOC)

To measure the invasiveness between model code and environmental modeling framework code, we counted the total number of lines of code which depend on the framework. A framework dependent line of code is defined as a line of code that requires the framework in order to compile. This implies that a framework dependent line of code contains at least one framework specific reference. For measurement purposes, FDLOC is a surrogate for quantifying boilerplate code. Boilerplate code is defined herein as sections of code that are included in many places with little or no alteration, but that are required in order to adapt model code to run under the framework of implementation. For counting purposes, it is difficult to define a boilerplate line of code precisely, because the classification requires a functional interpretation by the person or tool performing the count. A boilerplate line of code may not include any framework specific reference, but its existence in the model code is still required for framework adaptation. Conversely, framework dependent lines of code can be easily identified because of the strict requirement that they contain a reference to the framework. In this study, we calculated two variations of FDLOC: raw count and a percentage relative to the total lines of model code (% FDLOC). The raw count can be used to compare the number of lines of framework dependent code between model implementations. The percentage of FDLOC versus all LOC in a program is used to compare framework dependent code density across model implementations. In this study, FDLOC was computed manually by inspecting source code. It may be desirable for future analysis of large scale systems to develop a tool for collecting the FDLOC metric.

2.4. Code quality measures

Code quality tends to be an elusive property to quantify in software measurement because its definition varies greatly depending on the software requirements. As stated previously, we are interested in understanding the impact of framework invasiveness with regard to the non-functional quality attributes of model code such as maintainability, understandability, portability, and reusability. For this study, as a surrogate for measuring non-functional attributes of model code quality, we use three primary measures (Fenton and Pfleeger, 1997): 1) size, measured by counting lines of code (LOC); 2) complexity, measured by determining cyclomatic complexity; and 3) coupling, measured using efferent coupling (fan-out), afferent coupling (fan-in), and coupling between object classes (CBO). Cyclomatic complexity (CC) counts the number of linearly independent paths through a program's source code. This is a surrogate for measuring code complexity and has been a widely used measure in computer science. Coupling between object classes (CBO) is perhaps the most well-known coupling measure. Ideally we would have used this measure for all model language/framework implementations. However, not all of the models were coded in an object-oriented language as some were implemented using C and FORTRAN. To quantify coupling, we measured efferent (fan-out) and afferent (fan-in) coupling because these measures can be collected against against both procedural and object-oriented code. Efferent coupling is the number of classes which make reference to a class and can be thought of as the number of uses "outside" of the class. Afferent coupling is a dependency measure which counts the number of classes referenced by a class and can be thought of as the classes used "inside" the class. Previous research has suggested that large code size and high degrees of certain types of coupling inversely correlate with maintainability, an important non-functional quality attribute (e.g., Dagpinar and Jahnke, 2003; Anda, 2007).

2.5. Development/analysis tools

Static analysis tools that supported analysis of FORTRAN, C/C++, and Java were used to analyze the model implementations. SLOCCOUNT (Wheeler, 2009) was used to count lines of code. Understand 2.0 Analyst (Scientific Tools, 2009) was used to collect the LOC, CC, CBO, and fan-in/fan-out coupling software metrics. Function and data type usage reports produced by Understand 2.0 were parsed using a custom program to generate data for the FDT and FF usage measurements. FDLOC were determined manually by counting lines of code. The NetBeans integrated development environment (IDE) was used for all Java code development which included supporting the development of the OMS, OpenMI, and CCA based models. C/Fortran development was accomplished using a UNIX-based text editor.

3. Results

3.1. Thornthwaite model

For the Thornthwaite model, the framework implementations were coded to generate identical functionality and output given the same inputs. This approach allowed us to attribute differences observed between the model implementations to the differences among the various languages and frameworks used. The size and complexity measurements of the Thornthwaite model framework implementations are shown in Table 3. We observed a five-fold variation in model size from a low of 295 lines of code (LOC) for OMS 3.0 to a high of 1635 LOC for CCA. The OMS 3.0 framework was the only framework which enabled a smaller model (in LOC) than the implementation in the equivalent native language, i.e., the OMS 3.0 Thornthwaite implementation was 295 LOC compared to 319 for Java-only. Ideally, a framework-based model implementation should have a smaller code size than a plain-language implementation where the reduction in code size reflects code reuse from aspects of the model's functionality being provided by the framework. Not counting non-framework (language only) models, Table 3 shows a two-fold variation in average cyclomatic complexity (CC/method) from a low of 1.18 for OMS 2.2 to a high of 2.38 for OMS 3.0 and a nine-fold variation in total CC from a low of 31 for OMS 3.0 to a high of 276 for CCA. The CCA framework produced an unusually large amount of generated code. The Thornthwaite model code size as implemented in CCA was so large that it was treated as a statistical outlier (+/− 2 standard deviations) for nearly all metrics collected in this study. To compensate for the large quantity of boilerplate code in the CCA model implementation, we only counted lines of code for the eight Java component implementation files where model code was actually inserted. These eight files, were essentially the only files that were physically edited during model implementation. Any unedited files automatically generated by CCA were ignored for our measurements. This approach allowed the CCA Thornthwaite model metrics to fall within two standard deviations as it allowed a similar number of source files to be analyzed for the model, essentially one file per modeling component.

Table 3
Thornthwaite lines of code (LOC) and cyclomatic complexity (CC) metrics.

Language/Framework	Total LOC	Average CC/method	Total CC
FORTRAN only	244	3.33	40
OMS 3.0 Java	295	2.38	31
Java only	319	2.85	37
C++ only	405	2.41	41
OMS 2.2 Java	450	1.18	103
ESMF 3.1.1 C	583	1.97	65
ESMF 3.1.1 FORTRAN	683	1.44	56
OpenMI 1.4 Java	880	1.61	116
CCA 0.6.6 Java	1635	2.25	276

Coupling measures for the Thornthwaite model framework implementations are shown in Table 4. We observed nearly a two-fold variation in total fan-out (efferent coupling) from a low of 100 for ESMF C to a high of 195 for CCA and a three-fold variation in total fan-out (efferent coupling) from a low of 70 for OMS 2.2 to a high of 215 for CCA. For coupling between object classes (CBO), only four model implementations could be measured since CBO was not measurable for the C and FORTRAN implementations. Additionally, OMS 2.2 uses XML configuration files to specify all

Table 4
Thornthwaite model coupling measures. CBO is coupling between objects.

Language/Framework	Total Fan-In (Afferent)	Total Fan-Out (Efferent)	Average CBO/class
FORTRAN only	N/A	N/A	N/A
OMS 3.0 Java	116	70	0.89
Java only	67	92	0.89
C++ only	75	115	0.89
OMS 2.2 Java	116	70	0
ESMF 3.1.1 C	100	155	N/A
ESMF 3.1.1 FORTRAN	N/A	N/A	N/A
OpenMI 1.4 Java	126	177	1.1
CCA 0.6.6 Java	195	215	0

Table 5
Thornthwaite model framework invasiveness rankings (lowest to highest) by measure.

Framework Implementation	FDT-used	FDT-uses	FF-used	FF-uses	FDLOC
OMS 3.0 Java	1	1	2	1	1
OMS 2.2 Java	3	2	1	2	2
ESMF 3.1.1 FORTRAN	2	4	3	4	4
ESMF 3.1.1 C	5	5	4	3	3
OpenMI 1.4 Java	4	3	5	6	5
CCA 0.6.6	6	6	6	5	6

component interaction resulting in zero measured CBO for the model. For the Thornthwaite model framework implementations, measurements for size, complexity and coupling were positively correlated. Total LOC and CC had a correlation coefficient of $r = 0.94$ ($df = 4$, $p < 0.01$), total LOC and total fan-in had a correlation coefficient of $r = 0.92$ ($df = 3$, $p < 0.05$), and total CC with total fan-in had a correlation coefficient of $r = 0.95$ ($df = 3$, $p < 0.02$).

In this study, all frameworks with the exception of the OMS 3.0 framework can be classified as traditional frameworks as described in Section 1. These frameworks generally provide specialized data types to wrap native language data types, and also contain numerous API functions which are used to implement component definition and communication. Invasiveness measure rankings (trending from lowest to highest) for the Thornthwaite model framework implementations are shown in Table 5. The rankings in Table 5 are based on detailed results of the individual invasiveness measures listed in Table 6. For the framework invasiveness measures, the OMS 3.0 Thornthwaite model framework implementation appeared to be the least invasive, i.e., this implementation had far fewer framework dependencies than others. A 12-fold variation in FDLOC was observed from a low of 44 for OMS 3.0 to a high of 533 for CCA. We observed a 15-fold variation for framework data types used (FDT-used), from a low of 1 for OMS 3.0 to a high of 15 for CCA. For framework data type uses (FDT-uses), we observed a 135-fold variation from a low of 1 for OMS 3.0 to a high of 135 for CCA. Framework functions used (FF-used) varied seven-fold from a low of 7 for OMS 2.2 (OMS 3.0 had 8) to a high of 48 for CCA. Framework function uses (FF-uses) varied 13-fold from a low of 21 for OMS 3.0 to a high of 280 for CCA. The Thornthwaite scientific code was essentially the same for all of the environmental modeling framework implementations, with the observed differences resulting from various framework-specific requirements to implement the model. The large variations in the metrics suggest that variations in framework design likely impact the modeling code.

Table 6 also shows framework invasiveness metrics scaled to a percentage. The percentage scaling shows how much of the overall percentage of an attribute is framework dependent. For FDLOC (%), nearly a three-fold variation was observed from a low of 14.84% of LOC dependent on the framework for OMS 3.0 to a high of 41.42% LOC dependent on the framework for ESMF FORTRAN. For FDT-used (%), a ten-fold variation was observed from a low of 4.67% for OMS 3.0 data types used being framework dependent to a high of 46.88% for CCA. For FDT-uses (%), a 47-fold variation was observed from a low of 1.35% of OMS 3.0 data type uses being framework dependent to a high of 64.29% for OMS 2.2. For FF-used (%), a nearly three-fold variation was seen from a low of 26.67% of OMS 3.0 functions used being framework dependent to a high of 78.57% for ESMF FORTRAN. FF-uses (%) varied more than two-fold from a low of 40.38% of OMS 3.0 function uses being framework dependent to a high of 96.1% for ESMF FORTRAN. Overall, a model implementation with low framework invasiveness should have a low percentage of data type, functions, and LOC dependence on the underlying framework.

The final invasiveness measurement scaling shown in Table 6 is a scaling of attribute occurrences per 1000 lines of code (KLOC). Since the model implementations varied in size, this scaling provides a method for a side-by-side comparison. For FDLOC/KLOC, nearly a three-fold variation was observed from 148 for OMS 3.0 to 414 for ESMF FORTRAN. For FDT-used/KLOC, a five-fold variation was seen from 3.39 for OMS 3.0 to 17.15 for ESMF C. For FDT-uses/KLOC, a 61-fold variation was observed from 3.39 for OMS 3.0 to 209.26 for ESMF C. FF-used/KLOC was shown to exhibit nearly a two-fold variation from 15.56 for OMS 2.2 to 29.36 for CCA. FF-uses/KLOC was observed to have a three-fold variation from 216.69 for ESMF FORTRAN to 71.19 for OMS 3.0.

FDLOC, FDT-used, FF-used correlated with model size ($df = 4$, $p < 0.05$); however, none of the percentage or scaling invasiveness measures correlated with size. For complexity, three invasiveness measures (FDLOC, FDT-used, and FF-used) were shown to correlate

Table 6
Framework invasiveness detailed measurements for Thornthwaite.

Framework Implementation	FDLOC	FDT-used	FDT-uses	FF-used	FF-uses
OMS 3.0 Java	44	1	1	8	21
OMS 2.2 Java	147	5	72	7	33
ESMF 3.1.1 C	178	10	122	13	77
ESMF 3.1.1 FORTRAN	280	3	109	11	148
OpenMI 1.4 Java	338	8	73	20	280
CCA 0.6.6 Java	533	15	135	48	215
Framework Implementation	FDLOC (%)	FDT-used (%)	FDT-uses (%)	FF-used (%)	FF-uses (%)
OMS 3.0 Java	14.84	4.67	1.35	26.67	40.38
OMS 2.2 Java	32.67	41.67	64.29	50.00	73.33
ESMF 3.1.1 C	30.85	30.30	49.59	46.43	76.24
ESMF 3.1.1 FORTRAN	41.42	27.27	51.90	78.57	96.10
OpenMI 1.4 Java	38.41	23.53	32.30	37.74	79.10
CCA 0.6.6 Java	32.60	46.88	49.82	70.59	69.58
Framework Implementation	FDLOC/KLOC	FDT-used/KLOC	FDT-uses/KLOC	FF-used/KLOC	FF-uses/KLOC
OMS 3.0 Java	148	3.39	3.39	27.12	71.19
OMS 2.2 Java	327	11.11	160.00	15.56	73.33
ESMF 3.1.1 C	309	17.15	209.26	22.30	132.08
ESMF 3.1.1 FORTRAN	414	4.39	159.59	16.11	216.69
OpenMI 1.4 Java	384	9.09	82.95	22.73	318.18
CCA 0.6.6 Java	326	9.17	82.57	29.36	131.50

Table 7
PRMS model lines of code (LOC) and cyclomatic complexity metrics.

Framework Implementation	Total LOC	Average CC/method	Total CC
OMS 3.0 Java	10163	9.75	702
OMS 2.2 Java	16997	1.37	2575

with total CC (df = 4, $p < 0.05$). A correlation existed between FF-used/KLOC and average method cyclomatic complexity; however, correlation coefficients for other measures with average CC/method were almost random, so it is possible the FF-used/KLOC relation is incidental. Correlation coefficients between invasiveness and total complexity were generally positive though they varied in magnitude. Total fan-in (afferent) and fan-out (efferent) coupling correlated significantly with FDLOC, FDT-used, and also %FF-used (fan-in only) (df = 3, $p < 0.05$).

3.2. PRMS model

The invasiveness metrics were also applied to evaluate two Java-based PRMS model implementations under the OMS 2.2 and 3.0 frameworks. The size and complexity metrics for the PRMS framework implementations are shown in Table 7. PRMS model code size was reduced 40% in the OMS 3.0 framework implementation. Much of the size reduction can be attributed to the elimination of component getter and setter methods. “Getter” and “setter” methods are accessor methods which intercept read/write access to data variables in an object-oriented (OO) program. These constructs are encouraged as a way to provide data encapsulation, which is used to prevent unintentional changes to variables. Data encapsulation is an encouraged best practice in structured and OO programming. More recently it has been recognized that data encapsulation as a best practice is overused, particularly when getter/setter accessor methods never provide any additional functionality. In these cases, getters and setters should be removed to reduce the overall code size and should only be included when actual data encapsulation is being performed. The OMS 3.0 PRMS implementation makes use of plain old Java objects (POJOs) which use only native language data types, whereas the OMS 2.2 PRMS implementation uses framework-specific data types and framework-specific interfaces. The average complexity per method increased significantly from OMS 2.2 to OMS 3.0. This is because the total number of methods dropped significantly through elimination of the getter and setter methods. A reduction in model complexity is reflected in the more than three-fold reduction in total CC observed in the OMS 3.0 PRMS model implementation versus OMS 2.2 (Table 7).

Coupling measures for the PRMS model implementations under the OMS 2.2 and 3.0 frameworks are shown in Table 8. Reductions in both total fan-out (efferent) and total fan-in (afferent) coupling are observed in the OMS 3.0 PRMS implementation. Coupling was likely reduced in relation to the reduction in code size attributed by removing getter and setter methods. The average number of methods per component dropped from 85 to 3.6 in OMS 3.0. There was no measurable coupling between object classes (CBO) in the PRMS model because the OMS frameworks handle component interaction independently from the model source code. This feature enables OMS model components to be reused by simply plugging them into other models since the components are not coupled to each other through code dependencies. Framework invasiveness measures for the PRMS model implementations are shown in Table 9. A significant reduction is seen in the use of framework data types and functions using the OMS 3.0 lightweight framework implementation.

Table 8
PRMS model coupling measures. CBO is coupling between objects.

Framework Implementation	Total Fan-In (Afferent)	Total Fan-Out (Efferent)	Average CBO/class	Avg Number Methods/Class
OMS 3.0 Java	1232	755	0	3.6
OMS 2.2 Java	3517	1428	0	85.4

Table 9
PRMS model invasiveness measures.

Framework Implementation	FDT-used	FDT-uses	FF-used	FF-uses
OMS 3.0 Java	1	3	5	7
OMS 2.2 Java	16	1788	15	2854
Framework Implementation	FDT-used (%)	FDT-uses (%)	FF-used (%)	FF-uses (%)
OMS 3.0 Java	5	0.19	5.5	2
OMS 2.2 Java	50	65.9	19.2	91.8
Framework Implementation	FDT-used/ KLOC	FDT-uses/ KLOC	FF-used/ KLOC	FF-uses/ KLOC
OMS 3.0 Java	0.09	0.29	0.49	0.69
OMS 2.2 Java	0.94	105.2	0.88	167.9

3.3. Framework invasiveness and model size

One argument against using the Thornthwaite model to evaluate environmental modeling framework to code invasiveness is that the model code size is small and model functionality is not representative of larger-scale scientific models in wide use. One reason for PRMS model implementation and evaluation is the opportunity to obtain framework invasiveness data for a much larger model. We compared the Thornthwaite and PRMS model implementations using the framework invasiveness density measures from Tables 6 and 9 to examine if large, complex model implementations exhibited similar invasiveness compared with smaller, simplistic model implementations. Grouping the four framework invasiveness measures together, we found a general correlation between the two models for invasiveness density measures ($p = 0.012$, df = 6). The density of FDT-used was similar for the OMS 3.0 (4.67%, 5.0%) and OMS 2.2 (41.67%, 50%) frameworks for both the Thornthwaite and PRMS models. Density of FDT-uses was also similar for the OMS 3.0 (1.35%, 0.19%) and OMS 2.2 (64.29%, 65.9%) frameworks. The density of FF-used was shown to be lower for a larger-scale model with OMS 3.0 (26.67%, 5.5%) and OMS 2.2 (50.0%, 19.2%) for Thornthwaite and PRMS respectively. Finally, the density of FF-uses was found to not be similar for OMS 3.0 (40.38%, 2%) but similar for OMS 2.2 (73.33%, 91.8%) for Thornthwaite and PRMS respectively. Framework function usage density represents the ratio of framework function usage versus all function usage. In a larger, more complex model such as PRMS, we surmise that a greater quantity of the function usage results from the modeling logic itself and not simply calls to framework functions. Since Thornthwaite was a much smaller model overall, the majority of function usage was framework-based. A larger study which investigates additional models of varying complexity should help to provide a better understanding of the relationships between model size and framework invasiveness.

4. Discussion

This research sought to investigate the implications of framework invasiveness on non-functional attributes of model code quality and to explore the utility of the lightweight framework

approach for scientific and environmental modeling. Our initial investigation produced the following observations which are discussed below.

4.1. Framework invasiveness and model quality

To investigate relationships between non-functional attributes of model code quality and invasiveness, we used the Chidamber and Kemerer's (1994) object-oriented software metrics as indirect measures. Briand et al. (1999, 2000) identified coupling as an important dimension for predicting quality of object-oriented systems and coupling has also been shown to help to predict software failures in object-oriented code. Furthermore, size and coupling have been shown to be useful in predicting the maintainability of code. For this study, we lacked necessary data for coupling between object classes (CBO). Instead, we used efferent (fan-out) and afferent (fan-in) coupling measures. These measures were calculated for five different framework implementations of the Thornthwaite model (OMS 3.0, OMS 2.2, ESMF C, OpenMI, CCA), and two different framework implementations of the PRMS model (OMS 2.2 and 3.0). We observed that framework implementations for both models having the lowest invasiveness measures for FDT-uses and FF-uses also had the lowest values for fan-in/fan-out coupling ($p = 0.002$, df $= 5$; $p = 0.011$, df $= 5$). Framework implementations for both models with higher fan-in/fan-out coupling used more framework functions and data types. This relationship shows that the framework invasiveness measures strongly correlate with fan-in/fan-out coupling measures. We also found that model framework implementations with low invasiveness measures for FDT-uses and FF-uses had the smallest code sizes (LOC) ($p = 0.024$, df $= 5$; $p = 0.024$, df $= 5$) and total CC ($p = 0.0007$, df $= 5$; $p = 0.0007$, df $= 5$). Models with larger LOC and total CC used more framework functions and data types. This relationship shows that the framework invasiveness measures were correlated with both size and complexity measures. Previous research (e.g., Anda, 2007) has identified a negative relationship between both size and coupling with software maintainability while our research shows that framework invasiveness correlates with size and coupling. When considering prior research, this finding suggests a possible important connection between framework invasiveness and non-functional code quality attributes such as maintainability.

4.2. Environmental modeling using the lightweight framework approach

In this study, the OMS 3.0 framework with a lightweight framework design approach was utilized to produce model implementations for both the Thornthwaite and PRMS models. Five measures were applied to quantify framework to model invasiveness: FDLOC, FF-used, FF-uses, FDT-used, and FDT-uses. Using these measures, the OMS 3.0 model implementations had lower framework to model invasiveness (Tables 6 and 9) when compared with the other frameworks in this study. Additionally, the OMS 3.0 Thornthwaite and PRMS model implementations had lower overall code size (LOC), lower cyclomatic complexity (CC), and lower afferent (fan-in) and efferent (fan-out) coupling (Tables 4 and 8). Overall, the lightweight framework approach as tested using OMS 3.0 produced models which resulted in both smaller and simpler model implementations in terms of code size and complexity, and with lower invasiveness and coupling. These results highlight possible code quality improvements which may be realized using a lightweight framework approach to model implementation. Previous software engineering research suggests that lower coupling may result in fewer system faults (Briand et al., 1999, 2000; Dagpinar and Jahnke, 2003). Our investigation into the lightweight framework approach to modeling shows promise in helping to improve both functional and non-functional attributes of model code quality.

4.3. Study limitations

This study contained a number of limitations which should be pointed out:

- OMS 3.0 was the only framework classified as a lightweight framework. Ideally, our study would have included more than one lightweight framework but none were available to us.
- Code size (LOC), cyclomatic complexity (CC), and coupling are structural code measures and they do not directly measure code quality. Their usage herein has been as a surrogate for code quality. The relationships between code quality and these measures have been established by prior research in software measurement within the field of computer science. Future research might propose studies which more directly investigate relationships between framework invasiveness and specific non-functional code quality attributes such as maintainability. However it should be noted that the exercise of quantifying any non-functional quality attribute for such studies will always be a challenge.
- ESMF 3.1.1 did not support Java, thereby forcing us to compare C/FORTRAN Thornthwaite model implementations with Java-based implementations.
- The framework function (FF) and framework data type (FDT) invasiveness measures did not account for the use of non-traditional types of framework dependencies, such as the use of language annotations and XML configuration files (e.g., Guerra et al., 2009). Annotations are neither framework functions or data types so they are not accounted for with the FF/FDT metrics. However, the framework dependent lines of code (FDLOC) measure did count annotation lines of code as framework dependent lines. An enhancement to the FF/FDT measures might be to include some way to quantify the use of annotations as a framework dependency. One simple approach might be to simply count annotations as framework data type usages. However, it should be noted that when porting code to another language or framework annotations can easily be removed, similar to how programming comments can be removed. We posit that the impact of annotations on the refactoring effort is likely to be significantly less than the effort required to replace a framework specific data type or function usage.
- The researchers had to learn how to implement models using the CCA 0.6.6, ESMF 3.1.1 and OpenMI 1.4 frameworks for the Thornthwaite model implementations. Therefore, it is possible that the CCA, ESMF and OpenMI Thornthwaite model implementations were "non-ideal" due to initial inexperience with these frameworks. This limitation however is largely offset by the simplicity of the Thornthwaite model. To counteract this effect, both the ESMF and OpenMI based model implementations were submitted and reviewed by their respective framework designers. For each of our Thornthwaite implementations, significant time was spent by the researchers so that each model implementation was as concise and compact as possible. Since Thornthwaite was not a large or complex model, it is unlikely that an implementation by framework experts would differ significantly from ours.

5. Summary and conclusions

This paper presents a unique comparison of environmental modeling framework invasiveness using the Thornthwaite and

PRMS hydrologic models implemented using a variety of available frameworks. Framework invasiveness measures were developed to quantify three attributes: framework data type usage, framework function usage, and quantity of model code dependent on a framework. For the environmental modeling frameworks studied, less invasive model implementations exhibited an inverse correlation with code size, complexity and coupling. Models implemented using the OMS 3.0 framework had the lowest invasiveness numbers and also the smallest size, complexity, and coupling. For the Thornthwaite model, the OMS 3.0 implementation was only 40% as large as the average size of the traditional framework implementations and about 30% as complex. For the PRMS model, the OMS 3.0 implementation was 40% smaller and about 30% as complex as the traditional non-lightweight OMS 2.2 based framework implementation. Overall, the OMS 3.0 framework produced less invasive model implementations when compared to the traditional framework model implementations using OMS 2.2, ESMF 3.1.1, OpenMI 1.4, and CCA 0.6.6. In conclusion, our study showed that the use of a lightweight framework produced smaller, less complex models with less coupling and framework-to-model invasiveness leading us to suggest that a lightweight framework approach for environmental modeling frameworks deserves further attention.

Previous research has shown that structural measures are useful in predicting non-functional quality attributes of code. Our invasiveness measures were shown to correlate with some of these same structural measures. This suggests a potential relationship between invasiveness and non-functional quality attributes of model code. Based on our results, a lightweight framework approach to environmental modeling is suggested in order to better develop model code which is less dependent on a specific framework. The long-term implications of using lightweight frameworks should be studied, and future research should examine additional models as more lightweight frameworks and lightweight framework model implementations become available for study. In addition, more detailed studies to isolate the impact of invasiveness on specific non-functional code quality attributes are desirable. For example, an in-depth study could be designed to investigate more directly the relationship between invasiveness and a specific non-functional quality attribute such as maintainability. Further research and application of lightweight frameworks should help us to better understand the utility of this approach, ultimately guiding the environmental modeling community towards better framework designs.

Acknowledgments

We would like to acknowledge Cecelia DeLuca from UCAR, a member of the ESMF support team, who provided a review of the ESMF FORTRAN Thornthwaite model implementation. We would also like to thank Dr. Andrea Antonello who provided a code review of our OpenMI 1.4 Java Thornthwaite implementation.

References

Ahuja, L.R., Ascough II, J.C., David, O., 2005. Developing natural resource modeling using the object modeling system: feasibility and challenges. Advances in Geosciences 4, 29–36.

Anda, B., 2007. Assessing software system maintainability using structural measures and expert Assessments. In: Proceedings of the 23rd Intl. IEEE Conference on Software Maintenance (ICSM 2007), October 2–5, Paris, France, pp. 204–213.

Argent, R.M., 2005. A case study of environmental modelling and simulation using transplantable components. Journal of Environmental Modelling & Software 20 (12), 1514–1523.

Argent, R.M., Voinov, A., Maxwell, T., Cuddy, S.M., Rahman, J.M., Seaton, S., Vertessy, R.A., Braddock, R.D., 2006. Comparing modelling frameworks – a workshop approach. Journal of Environmental Modelling & Software 21 (7), 895–910.

Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S., Smolinski, B., 1999. Toward a common component architecture for high-performance scientific computing. In: Proceedings of the 8th IEEE Intl. Symposium on High Performance Distributed Computing (HDPC '99), August 3–6, Redondo Beach, CA, USA, pp. 115–124.

Basil, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22 (10), 751–761.

Blind, M., Gregersen, J.B., 2005. Towards an open modeling interface (OpenMI) the HamonET project. Advances in Geosciences 4, 69–74.

Briand, L.C., Wiist, J., Ikonomovski, H.L., 1999. Investigating quality factors in object-oriented designs: an industrial case study. In: Proceedings of the 21st International Conference on Software Engineering (ICSE '99), May 16–22, Los Angeles, CA, USA, pp. 345–354.

Briand, L.C., Wust, J., Daly, J., Porter, D.V., 2000. Exploring the relationships between design measures and software quality in object-oriented systems. Journal of Systems & Software 15 (3), 245–273.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. Transactions on Software Engineering 20 (6), 476–493.

Collins, N., Theurich, G., DeLuca, C., Suarez, M., Trayanov, A., Balaji, V., Li, P., Yang, W., Hill, C., Silva, A., 2005. Design and implementation of components in the Earth system modeling framework (ESMF). International Journal of High Performance Computing Applications 19 (3) Fall/Winter 2005.

Dagpinar, M., Jahnke, J., 2003. Predicting Maintainability with Object-Oriented Metrics – An Empirical Comparison. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03), November 13–16, Victoria, B.C., Canada, pp. 155–164.

David, O., Markstrom, S.L., Rojas, K.W., Ahuja, L.R., Schneider, W., 2002. The object modeling system. In: Ahuja, L.R., Ma, L., Howell, T.A. (Eds.), Agricultural System Models in Field Research and Technology Transfer. Lewis Publishers, Boca Raton, FL, USA, pp. 317–344.

David, O., Ascough II, J., Leavesley, G., Ahuja, L.R., 2010. Rethinking modeling framework design: object modeling system 3.0. In: Swayne, Yang, Voinov, Rizzoli, Filatova (Eds.), iEMSs 2010 International Congress on Environmental Modeling and Software – Modeling for Environment's Sake, Fifth Biennial Meeting, July 5–8, Ottawa, Canada, p. 8.

Dig, D., Johnson, R., 2006. How do APIs evolve? A story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice 18, 83–107.

Donatelli, M., Rizzoli, A.E., 2008. A design for framework-independent model components of biophysical systems. In: Sànchez-MarrèBéjar, Béjar, Comas, Rizzoli, Guariso (Eds.), iEMSs 2008 International Congress on Environmental Modeling and Software – Modeling for Environment's Sake, Fourth Biennial Meeting, July 7–10, p. 9. Barcelona, Catalonia.

Elrad, T., Filman, R., Bader, A., 2001. Aspect-oriented programming: introduction, 2001. Communications of the ACM 44 (10), 28–32.

Fenton, N., Pfleeger, S., 1997. Software Metrics: A Rigorous & Practical Approach, Second Ed. PWS Publishing Company, Boston, MA.

Fowler, M., 2004. Inversion of Control Containers and the Dependency Injection Pattern. 19 p.. Available from. http://www.martinfowler.com/articles/injection.html (accessed 01.11).

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA.

Guerra, E.M., Silveira, F.F., Fernandes, C.T., 2009. Questioning traditional metrics for applications which uses metadata-based frameworks. In: Proceedings of the 3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM '09), October 26, Orlando, Florida, USA, pp. 35–39.

Jagers, H.R.A., 2010. Linking data, models and tools: an overview. In: Yang, Voinov, Rizzoli, Filatova (Eds.), IEMSS 2010 International Congress on Environmental Modeling and Software – Modeling for Environment's Sake, Fifth Biennial Meeting, July 5–8. Swayne, Ottawa, Canada, p. 8.

Leavesley, G.H., Lichty, R.W., Troutman, B.M., Saindon, L.G., 1983. Precipitation-Runoff Modeling System (PRMS) User's Manual U.S. Geological Survey water resources investigation Report 83-4238, 207 p.

Leavesley, G.H., Markstrom, S.L., Restrepo, P.J., Viger, R.J., 2002. A modular approach to addressing model design, scale, and parameter estimation issues in distributed hydrological modelling. Hydrological Processes 16 (2), 173–187.

Leavesley, G.H., Markstrom, S.L., Viger, R.J., 2006. USGS modular modeling system (MMS) - precipitation-runoff modeling system (PRMS). In: Singh, V.P., Frevert, D.K. (Eds.), Watershed Models. CRC Press, Boca Raton, FL, pp. 159–177.

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. Journal of Systems and Software 23 (2), 111–122.

Markstrom, S.L., Niswonger, R.G., Regan, R.S., Prudic, D.E., Barlow, P.M., 2008. GSFLOW-Coupled Ground-water and Surface-water FLOW Model Based on the Integration of the Precipitation-Runoff Modeling System (PRMS) and the Modular Ground-Water Flow Model (MODFLOW-2005) U.S. Geological Survey Techniques and Methods 6–D1, 240 p..

Maxwell, T., 1999. A paris-model approach to modular simulation. Journal of Environmental Modelling & Software 14 (6), 511–517.

McCabe, G.J., Markstrom, S.L., 2007. A Monthly Water-Balance Model Driven by a Graphical User Interface USGS Open-File Report 2007–1088, 6 p..

Mitchell, J.C., 1996. Foundations of Programming Languages. MIT Press., Cambridge, MA.

Carlson, J., 2010. Object Modeling System. Available from. http://oms.javaforge.com (accessed 01.11).

Rahman, J.M., Seaton, S.P., Perraud, J.M., Hotham, H., Verrelli, D.I., Coleman, J.R., 2003. It's TIME for a new environmental modelling framework. In: Post, D.A. (Ed.), MODSIM 2003 International Congress on Modelling and Simulation. Modeling and Simulation Society of Australia and New Zealand, July 2003, pp. 1727–1732.

Rahman, J.M., Seaton, S.P., Cuddy, S.M., 2004. Making frameworks more useable: using model introspection and metadata to develop model processing tools. Journal of Environmental Modelling & Software 19 (3), 275–284.

Reed, M., Cuddy, S.M., Rizzoli, A.E., 1999. A framework for modelling multiple resource management issues – an open modelling approach. Journal of Environmental Modelling & Software 14 (6), 503–509.

Richardson, C., 2006a. POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks. Manning Publications Co., Greenwich, CT.

Richardson, C., 2006b. Untangling enterprise Java. ACM Queue 5 (4), 33–44.

Rizzoli, A.E., Davis, J.R., Abel, D.J., 1998. Model and data integration and re-use in environmental decision support systems. Decision Support Systems 24, 127–144.

Scientific Toolworks, 2009. Understand – Source Code Analysis and Metrics. Available from. Scientific Toolworks, Inc. http://www.scitools.com (accessed 01.11).

SpringSource, 2010. SpringSource.org, a division of Vmware. http://www.springsource.org Available from(accessed 01.11).

Tapestry, Apache, 2010. Apache Tapestry – Welcome to Tapestry. Available from. Apache Software Foundation. http://tapestry.apache.org (accessed 01.11).

Terracotta, 2010. Terracotta. Available from. http://www.terracotta.org (accessed 01.11).

Thornthwaite, C.W., 1948. An approach toward a rational classification of climate. Geographical Review 38 (1), 55–94.

Voinov, A., Costanza, R., Wainger, L.A., Boumans, R.M.J., Villa, F., Maxwell, T., Voinov, H., 1999. Patuxent landscape model: integrated ecological economic modeling of a watershed. Journal of Environmental Modelling & Software 14 (5), 473–491.

Voinov, A., Fitz, C., Boumans, R., Costanza, R., 2004. Modular ecosystem modeling. Journal of Environmental Modelling & Software 19 (3), 285–304.

Watson, F.G.R., Rahman, J.M., 2004. Tarsier: a practical software framework for model development, testing and deployment. Journal of Environmental Modelling & Software 19 (3), 245–260.

Wheeler, D.A., 2009. SLOCCount. http://www.dwheeler.com/sloccount/ (accessed 01.11).