Using Server Pages to Unify Clones in Web Applications: A Trade-off Analysis

Damith C. Rajapakse and Stan Jarzabek
Department of Computer Science
School of Computing, National University of Singapore
{damith,stan}@comp.nus.edu.sg

Abstract

Server page technique is commonly used for implementing web application user interfaces. Server pages can represent many similar web pages in a generic form. Yet our previous study revealed high rates of repetitions in web applications, particularly in the user interfaces. Code duplication, commonly known as ‘cloning’, signals untapped opportunities to achieve simpler, smaller, more generic, and more maintainable web applications. Using PHP Server page technique, we conducted a case study to explore how far Server page technique can be pushed to achieve clone-free web applications. Our study suggests that clone unification using Server pages affects system qualities (e.g., runtime performance) to the extent that may not be acceptable in many project situations. Our paper discusses the trade-offs we observed when applying Server pages to achieve simple, non-redundant design in a web applications. We expect our findings to help in developing and validating complementary techniques that can unify clones without incurring such trade-offs.

1. Introduction

Repeated similar program structures (aka ‘clones’) make the code base larger than necessary. They hinder program comprehension by injecting implicit dependencies among program parts. Tracing and updating all the clones is a tedious and error-prone process, often resulting in update anomalies. Unifying clones with unique generic representations reduces the code size, explicates the dependencies, and reduces the chance of update anomalies. In industrial projects [16][26] and lab studies [25], we observed that using suitable generic design techniques we can unify many clones, achieving high reuse rates and productivity gains during development and maintenance.

Yet clones continue to plague today’s software. We analyzed 17 web applications (WAs) of different sizes, developed using different technologies, in different application domains [20]. We found a high incidence of clones, particularly in the user interface (UI) area.

Server page (aka dynamic page generation) techniques are commonly used for implementing WA UIs. ASP, JSP, and PHP are typical examples for web technologies that use some form of dynamic page generation. In essence, a Server page contains a combination of HTML and programming language scripts, and the web server uses it to generate web pages at runtime. A Server page can represent many similar web pages in a generic form, providing an alternative to cloning. But how far this capability can be pushed to achieve clone-free WAs is an intriguing and useful question yet to be answered.

In this paper, we present the trade-offs we encountered when we attempted to use Server page technique to unify clones in a WA. Our analysis is based on a case study involving alternative designs of a WA called Project Collaboration Environment (PCE). We built PCE based on requirements from one of our industry projects [16]. We selected the Server page technique of PHP to implement PCE. We incrementally applied design patterns and PHP features to unify clones in PCE, progressively replacing clones with generic representations. We did three consecutive implementations of PCE, where each implementation was a refinement of the previous one.

As we moved from the first implementation to the third, we were able to unify most of the clones that were significant enough to justify the effort. This resulted in a significant reduction of the code size (by 78%), and a lesser chance of update anomalies (number of modification points dropped from 251 to 8 for certain changes).

Throughout the experiment, we analyzed how unifying clones in PCE was affecting other engineering qualities of the PCE. We observed that clone unification caused many trade-offs. While some of these tradeoffs were well known in traditional software development, the majority of them were less obvious, and applicable only in the context of WA
development. These trade-offs resulted from the interplay between clone reduction, realities of WA development (such as fuzzy requirements, dramatically short development schedules, constant evolution, and shortened revision cycles [3]), and desirable engineering qualities of WAs (such as high performance, high information content, and good aesthetics). We believe that some of these trade-offs would be unacceptable in many WA development situations.

Detail analysis and description of these trade-offs in both qualitative and quantitative terms is the main contribution of our paper. Our findings can help WA developers make more informed decisions when to (and when not to) apply Server page techniques. It follows from the study, that we need complementary methods that would allow us to avoid unnecessary clones without compromising other important properties of WAs. We expect our findings to help in the development and validation of such methods.

The rest of the paper is organized as follows. In Section 2 we give an overview of our experiment method. Section 3 gives the details of PCE and the three alternative implementations of PCE we did, with a comparison of size and cloning level at the end. In Section 4 we describe various trade-offs caused by clone unification. Section 5 briefly comments about the applicability of our observations to the J2EE and .NET platforms. After related work (Section 6), we present our conclusions and intended future work in Section 7.

2. Experiment method

We selected PHP as our Server page technology. PHP is a free, popular (20 million web domains used PHP by the end 2005 [17]), and versatile technology specifically geared for WA development. Although PHP started out as a simple scripting language, today it has evolved into an industrial strength WA technology, used by complex WAs such as sourceforge.net.

The experiment involved Project Collaboration Environment (PCE), based on requirements from our industry partner. We based PCE on CPG-Nuke [4] open source web portal. CPG-Nuke is an adaptation of PHP-Nuke [18], a popular open source WA, averaging ½ million downloads per year for last three years. As our focus was on the UI layer of PCE, we kept the other layers (i.e., business logic and database layers) simple.

We carried out three different implementations of PCE (see Figure 1), each one functionally equivalent to the other two. The first implementation was based on a very simple design, without much effort to minimize clones. We call this PCE\textsubscript{simple}. In the second implementation, named PCE\textsubscript{patterns}, we tried to unify clones by applying suitable design patterns to PCE\textsubscript{simple}. In PCE\textsubscript{unified}, we unified any remaining clones that, in our judgment, were worth this effort.

![Figure 1. The three PCE implementations](image_url)

In the experiment, we considered as ‘clones’ code structures that displayed enough textual similarity. A ‘code structure’ could be as small as a HTML/PHP code fragment, as large as a whole module, or anything in between such as a collection of functions/files. We considered clones as ‘worth unifying’ if they were of considerable length (at least 30 tokens long) and if their unification deflates the code involved by at least 25% (i.e., unified code is at least 25% smaller than the size of total clone instances). We used ‘CCFinder’ clone detection tool [12] and ‘Gemini’ clone visualization tool [24] to aid us in identifying and analyzing clones.

We categorized clones into intra-module clones (clones occurring within the same module) and inter-module clones (clones occurring across modules). In PCE\textsubscript{patterns}, we focused on intra-module clones, as they are more localized and easier to tackle. In PCE\textsubscript{unified} we widened our focus to cover intra-module clones as well. Once clones were selected for unification, we used a combination of following strategies to unify them. We give concrete examples of each strategy in Section 3.

- Applying design patterns – Some clones were unified by applying a design pattern that aims to reduce code duplication. We selected the suitable design pattern by comparing the clone characteristics against design patterns drawn from industry best practices in J2EE [1], .NET [23], and from platform-independent recommendations [11][9].
- Applying known refactoring techniques – Some clones were unified by applying commonly known refactorings, such as given in [10].
- Context-specific structurings – When a clone did not fit into a known design pattern or a refactoring, we applied PHP features and context-specific design/code restructuring to unify the clones.

Since our study was a controlled lab experiment, a question arises as to what extent our findings would be relevant to the industry practice. We tried to mitigate this shortcoming in the following ways:

- We used functional requirements and the conceptual model of a real WA built by our industry partner. We followed design best practices from industry (captured by design patterns) in PCE design.
- We reused industry accepted architectures and frameworks as the core of our PCE implementation.
• We maintained a tight feedback loop with our industry partner throughout the experiment to validate our findings.
• Finally, the three implementations were done by the first author who is a trained software engineer having prior industry experience in WA building.

In the end, the size and the cloning level of PCE were also comparable to a similar WA built by our industry partner [16].

3. Details of the experiment

3.1 Project Collaboration Environment (PCE)

PCE supports project record keeping, task assignment to staff, task progress tracking, and a range of other activities related to project planning and execution. It has six modules corresponding to six entity types in PCE domain, namely Staff, Project, Product, Task, Notes, and File. PCE maintains records of those entities and relationships among them. For example, Staff module maintains records of staff members, Product module tracks the status of project deliverables, Staff and Project modules maintain info about which staff members belong to which project teams, and Task and Staff modules maintain info about project tasks assigned to staff members. A screenshot from the Staff module given in Figure 2 shows a listing of staff members.

Figure 2. A screenshot from the Staff module

Figure 3 depicts main PCE entity types and relationships among them. Figure 4 depicts similarities and variations between PCE modules as a feature diagram [13]. A typical module M in PCE has a name (e.g., 'Staff'), and a number of attributes (e.g., Staff module has attributes Full Name, Job Title, ... cf Figure 4). Some attributes are common to all modules, while others – optional – are module-specific. Each module supports actions (create, edit, delete, ...). Some actions are further divided into sub-actions, some of which are optional. A module may optionally have relationships (association, and either 'strong' composition or 'weak' composition - but not both).

Figure 3. Domain model of PCE

Figure 4. Feature diagram of a PCE module

Figure 5 shows a PCE architecture, which was common to all three PCE implementations. The Foundation part of PCE consists of Admin Modules (used for administration of PCE) and Service Modules (used to provide various infrastructure services like database connectivity, logging, etc.). Foundation acts as a platform on which we deploy various User Modules. It provides a framework for implementing modules, and administration facilities to manage those modules. General User Modules provide common facilities to users (e.g., polls, message boards, preference management, etc.).
For the Foundation and General User Modules, we reused CPG-Nuke code as is whenever possible, and with minimal changes when necessary. The six PCE modules were deployed as another set of User modules. The reuse of CPG-Nuke reduced the PCE implementation to just these six modules. We built them in conformance with the Foundation requirements, so that they too could use Foundation services, and could be managed using the Foundation (e.g., we used the Foundation services for implementing a common look and feel). With the reuse of CPG-Nuke we hoped not only to reduce the implementation workload, but also to ensure that our implementation was based on an industry-accepted architecture.

3.2 PCE\textsubscript{simple}: a first-cut solution

We followed a so-called KISS principle (i.e., \textit{Keep It Simple, Straight-forward}) when implementing PCE\textsubscript{simple}. This initial version of PCE exemplifies a first-cut solution that is likely to emerge when developing a new WA under time pressure. The priority was to ‘get PCE done’, with maintainability concerns such as ‘clone avoidance’ taking a low priority. Each action (or sub-action) of the module in PCE\textsubscript{simple} was implemented as a single independent Server page, as shown in Figure 6. For example, createStaff.php page implemented the \texttt{create} action for \texttt{Staff} module. Cloning was liberally used when dealing with intra/inter-module similarities. For example, we implemented one module and used it to implement other modules by simply cloning it. Two forces heavily influenced the design of PCE\textsubscript{simple}:
1. Architectural guidelines implied by the Foundation – although our design was the simplest possible, we still adhered to the guidelines implied by the Foundation.
2. Conceptual design of a similar WA implemented by our industry partner [16] (source code was not available as it was a commercial application). PCE conceptual model (Figure 3), direct mapping of modules to entities, and the page-per-action organization in PCE\textsubscript{simple} were direct results of this. With the above two, we expected our PCE to closely match an industrial implementation.

3.3 PCE\textsubscript{patterns}: a pattern-based solution

The objective of PCE\textsubscript{patterns} was to reduce cloning in PCE\textsubscript{simple} by applying design patterns. We first re-organized our design around the \textit{Model-View-Controller} (MVC) pattern that is widely used for UI intensive applications. As per this pattern, each PCE entity consisted of a \textit{Model}, a number of \textit{Views}, and a number of \textit{Controllers} that updated the model and selected the appropriate View to visualize the Model. This is depicted in the top half of the meta-model of a PCE Entity shown in Figure 7.

Figure 8 shows a module designed by following this meta-model. Application of MVC was not meant to affect the cloning level directly. However, this was a necessary precursor to applying other design patterns that unify clones, since those patterns are targeted for an MVC architecture.
Then, we applied design patterns to unify identified intra-module clones. We applied these patterns within the scope of a module, repeatedly applying the same patterns to each module. Some examples of clone situations we found and the matching patterns selected are given next (the rest is omitted for brevity):

- Similar preprocessing sequences were repeated for each page request (e.g., session validation, parameter decoding). We applied the Front Controller [1][9][23] pattern to unify such clones into a single location. As per this pattern, each module has one Front Controller that receives all user requests and performs control tasks common to all requests (FrontController in Figure 7, StaffFrontController in Figure 8).

- Some views exhibited much similarity among them. We applied the Template View [9] pattern to unify such clones. That is, we put the similar parts of views into a template (ActionView in Figure 7, createStaffView in Figure 8), and used that template to generate various different cloned views it unifies.

- Data retrieval code was cloned in multiple views. We used the View Helper [1] pattern to unify the cloned code into a common helper class. Accordingly, some PCE Views are aided by a helper classes (ActionViewHelper in Figure 7, createStaffViewHelper in Figure 8).

- Some fragments of UI recurred in multiple places (e.g., attribute display code was cloned in Edit page as well as in Display page). Following the Composite View pattern [23], we unified such fragments into a smaller view that was then reused to compose larger views.

As mentioned earlier, we kept the non-UI parts of PCE as simple as possible; each module has minimal domain logic and is represented in the database as a single table. As recommended by [9] for such situations, we used the Table Module pattern and the Table Data Gateway patterns to this portion (i.e., Table module and TableDataGateway in Figure 7, omitted in Figure 8). In the controller portion, we also used the Page Controller [9][23] pattern to control the complexity of controllers. As per this pattern, each Front Controller uses a number of Page Controllers, one per each action supported by the module (ActionController in Figure 7, createStaffController in Figure 8), rather than have a single controller for all the actions.

3.4 PCEunified: further clone unification

PCEunified was an all-out effort to unify any remaining clones. First, we identified remaining intra-module clones in PCEpatterns and unified them using a combination of the following techniques:

- We extracted duplicated code fragments into methods using ‘extract method’ refactoring [10].
- We unified similar functions handling variations with ‘add parameter’ refactoring [10], conditional branches, and Template Method pattern [11].
- We converted similar HTML fragments to PHP Server pages, using PHP scripts to handle variations in HTML clones (an example of a ‘context-specific restructuring’ using PHP).
- further, more intensive, application of Composite View pattern to unify common parts of Views.

After the intra-module clones were dealt with, we shifted our focus to inter-module clones. Our clone detection indicated that there were enough inter-module clones to consider each whole module as a coarse-grain clone of the others. This was not surprising since modules initially implemented were used as blueprints for later modules. To remedy this situation, we unified the six modules into one generic module in the following manner: We pulled the six Front Controllers out of the modules and unified clones among them by creating two layers of Controllers. The top layer consisted of a common Front Controller that unified common control tasks. The second layer consisted of six module-specific controllers (e.g., StaffFrontController, ProjectFrontController, … in Figure 9). The rest of the six modules were unified into one module (called ‘unified module’ in Figure 9). Variations found were handled using the same techniques that we used to handle inter-module variations.
3.5 Overall comparison

We start by comparing the size and the cloning level in these three implementations of PCE. To measure the cloning level, we use the percentage of non-unique (i.e., cloned) code, calculated based on clones detected by CCFinder tool [12]. This measure is directly related to the probability of update anomalies. For instance, if 35% of the system is non-unique, any change to that 35% of the system risks an update anomaly. To minimize distortions created by false-positives and trivially short clones, only exact duplicates that are longer than 20 tokens were counted as clones. The details of this calculation can be found in [20].

Table 1 summarizes the cloning percentage (C%), LOC count, and number of files (#F), calculated for a typical module (we chose Project module as the typical module because it was used as the blueprint for other modules), and for all modules. The last column shows the inter-module cloning level (we chose Project module and Product module to calculate this metric). These data indicate a very high (98%) overall cloning level in PCEsimple, i.e., almost all code in PCEsimple is repeated in more than one place. This is because we copied existing modules to create new modules, resulting in many inter-module clones. This number is also comparable with findings of our industry case study [16], which reported that up to 90% of a new module may be implemented by reusing code from existing modules.

Table 1. Size and cloning level comparison

<table>
<thead>
<tr>
<th></th>
<th>Project module</th>
<th>All modules</th>
<th>Intermodule</th>
</tr>
</thead>
<tbody>
<tr>
<td>C% LOC #F #L</td>
<td>C% LOC #F #L</td>
<td>C% LOC #F #L</td>
<td></td>
</tr>
<tr>
<td>PCEsimple</td>
<td>55 1085 10 90</td>
<td>50 44 55</td>
<td></td>
</tr>
<tr>
<td>PCEpatterns</td>
<td>32 931 21 86</td>
<td>50 95 120</td>
<td></td>
</tr>
<tr>
<td>PCEunified</td>
<td>15 838 20 26</td>
<td>1120 32</td>
<td></td>
</tr>
</tbody>
</table>

We also see a noticeable drop in intra-module cloning from PCEsimple to PCEpatterns (from 55% to 15%). This shows that application of patterns has indeed reduced the cloning level. However, the repeated application of same patterns for each module has maintained the level of inter-module clones (cf last column of Table 1), and the overall cloning levels still high. Further unification of intra-module clones, followed by unification of modules has reduced both intra-module and overall cloning levels in PCEunified. A manual examination revealed that the remaining clones in PCEunified are either too small to warrant unification, or not practical to unify (Section 4.3 gives an example).

How does this affect maintainability? First, there is a significant drop in the size of code to be maintained. There is a 33% reduction in code size (in terms of LOC) within a module, from PCEsimple to PCEunified.

The overall system size has dropped much more (by 78%) largely due to unification of six modules into one. Second, the chance of update anomalies has reduced. Table 2 shows the distribution of the impact of three hypothetical evolutionary changes when carried out for one module, or for all modules. It illustrates how the number of modified files (#F) and modified locations (#L) decreases from PCEsimple to PCEunified, reducing the chance of an inconsistency during the update.

Table 2. Change propagation comparison

| Change 1 | 1 7 49 5 35 2 6 |
| Change 2 | 1 3 6 1 2 1 2 |
| Change 3 | 1 9 9 1 1 1 1 |

Change 1. Link all attribute names to a Glossary page.
Change 2. Move 'last edited time' to another location.
Change 3. Record each request to PCE in a log file.

4. Trade-off analysis

Figure 10 shows how the cloning level decreases as we go from PCEsimple to PCEunified. However, there are many other ways to design PCE, and the implementation of PCE in a given real production environment could land anywhere in this axis.

In our experiment, we observed how clone unification can lead to trade-offs in other WA properties that often should not be compromised. Such trade-offs can push the final result towards the left. This section describes these trade-off situations in detail. For each such situation, we discuss the WA engineering realities that set the context for the trade-off, and give concrete examples from PCE to illustrate how clone unification forces the trade-off.

4.1 Performance

Some of the WAs operate in the highly competitive environment of the Internet. As slower performance can drive users/customers away, ‘criticality of performance’ is one important characteristic for such WAs [5]. Unfortunately, clone unification can affect performance negatively by introducing additional function calls, function parameters, and ‘include’ directives. As an example, a simple comparison of page generation time for five randomly selected pages of Staff module is shown in Figure 11 (all other things...
being equal, averaged over 10 page requests, when PCE was hosted on a Pentium IV, 3GHz machine having 1 Gb memory). In all cases, page generation times of the three PCEs followed the pattern: \text{PCE}_{\text{simple}} < \text{PCE}_{\text{patterns}} < \text{PCE}_{\text{unified}}. On average (see last column), \text{PCE}_{\text{unified}} is more than three times slower than \text{PCE}_{\text{simple}}. This example shows how clone unification, although feasible, can incur performance trade-offs.

4.2 WYSIWYG editor compatibility

Three important characteristics of a WA are ‘aesthetics’, ‘information content’, and ‘constant evolution’ [5]. Therefore, the creation and maintenance of WA UI require continuous involvement of multimedia authors (e.g., graphic designers), content authors (e.g., technical writers), and programmers. The first two categories typically prefer to work with WYSIWYG authoring tools. Overzealous clone unification however, can interfere with such WYSIWYG editing. For example, the PCE UI was constructed as an HTML based template, and the program logic was placed in helper classes. Typically, a graphic designer creates the UI template using a WYSIWYG editor (e.g., Macromedia Dream Weaver), while a programmer builds helper classes. ‘Hooks’ (very short PHP scripts) in the template are used to extract the dynamic parts from the helper class. Except during the time programmer places hooks in the template, both experts work in parallel. We observed that intensive clone unification in \text{PCE}_{\text{unified}} had a negative impact on this setup. It brought more programming logic into the template (in the form of extra parameters, conditional branches, function calls), fragmented the template (e.g., when using Composite View pattern), and made rendering of the WYSIWYG editor increasingly different from the actual result. This shows that clone unification can force a trade-off in the ability of the WA UI to be edited using WYSIWYG editors.

4.3 Platform/framework conformance

It is typical to build WAs by using available platforms/frameworks, rather than build from scratch. However, each such platform/framework has conformance requirements. For instance, some of them require certain code/file to be physically present in a given location. We encountered two such examples in our experiment:

1. PCE Foundation required a certain security check to be placed at the beginning of each file, to prevent direct access to it.
2. PCE Foundation required each module to be in a separate folder (bearing the same name as the module), and a file named ‘index.php’ to be present in each such folder.

Clone unification can interfere with such physical presence requirements. In the first example, the unification of the clone would have prevented us from using the built-in security mechanism of the Foundation. In the second example, we modified the Foundation (generally a risky, undesirable option) to remove that requirement. These examples show how clone unification can force a trade-off in our ability to utilize platforms/frameworks when building WAs.

4.4 Ease of indexing by search engines

Success of some WAs depends on how easy it is for search engines to index them (e.g., e-commerce web sites). Clone unification using Server pages increases the amount of dynamic code in the WA UI. Since dynamic contents are less likely to be indexed by search engines, clone unification can force a trade-off in the WA’s ability to get indexed by search engines. A good example is an e-commerce application preferring not to unify cloned static pages in its product catalog.

Note: This trade-off is not directly related to PCE experiment. It was pointed out by one of our industry collaborators, based on their own experience in building e-commerce product catalogs.

4.5 Ability to use of multiple content types

While applications written in several languages are certainly nothing new, multilingualism is taken to a new level in WA development [22]. WAs are implemented using a mixture of content types (ASP, C#, CSS, DTD, HTML, Java, JavaScript, etc.). In our previous study [20], we found 59 content types in 17 WAs (we considered all text files that are likely to be maintained by hand); on average, one WA involved 10 different content types. Furthermore, some clones can involve multiple content types intertwined with each other. To give an example from PCE, two cloned files can include HTML, PHP, Java Script, and SQL. While each content type may have its own clone unification facilities, intermixing of multiple content types complicates clone unification. Therefore, a drive towards a high level of clone unification can force a trade-off in the ability to mix content types in a WA implementation.
4.6 Rapid development capability

Our experiment started with a clone-ridden implementation (i.e., PCE\textsubscript{simple}), and progressively unified clones to arrive at a clone-free implementation (PCE\textsubscript{unified}). But in a production environment we would prefer to achieve PCE\textsubscript{unified} as our first implementation, rather than go through three iterations. Clone unification is implicit in such a scenario. That is, clones are unified before they are created at all (in other words, ‘clone avoidance’). We can extrapolate our observations in ‘unifying’ clones to show that such ‘avoiding’ clones in an initial implementation too can incur a trade-off in another important property of a WA, as we shall explain next.

Being the ‘first-in-the-market’ can be a significant advantage for a commercial WA. Consequently, WAs have ‘compressed development schedules’ [5] (typically, less than 3 months [15]). However, clone avoidance requires additional effort, which may not be affordable for a WA project done under a compressed schedule. A comparison of the three PCE designs supports this argument; there are additional concepts, more indirection, and more layers as we go from PCE\textsubscript{simple} to PCE\textsubscript{unified} requiring more initial planning, analysis, and modeling. Therefore, despite the drop in LOC, the upfront development effort and time-to-market increases as we go from PCE\textsubscript{simple} to PCE\textsubscript{unified}. Although PCE\textsubscript{unified} is the smallest of the three, it is unlikely that we could have achieved the same high degree of clone unification in the first attempt, within the same time it took us to develop PCE\textsubscript{simple}. This shows that intense clone avoidance can force a trade-off in the ability to quickly release a working WA.

4.7 Rapid evolution capability

WA projects typically start with ‘insufficient requirement specification’ [5], and continuously have to evolve to match volatile requirements/technologies. This requires WAs to evolve rapidly. However, high level of clone unification can force a trade-off in this ability. This line of argument may appear to contradict Section 3.5, in which we illustrated how the number of modified files/locations decreases as we unify more clones (cf Table 2). This is not so, as we shall illustrate with the following example.

<table>
<thead>
<tr>
<th>Table 3. Effort for adding ‘strong composition’</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_modified</td>
</tr>
<tr>
<td>PCE\textsubscript{simple}</td>
</tr>
<tr>
<td>PCE\textsubscript{pattern}</td>
</tr>
<tr>
<td>PCE\textsubscript{unified}</td>
</tr>
</tbody>
</table>

Let us consider the effort required to add a new feature to the three PCEs. Table 3 shows what’s involved in adding the ‘strong composition’ feature to only one of the modules (assuming it only supported ‘weak composition’ to begin with), namely the number of files that may be affected by this new feature, number of files actually modified, number of independent locations modified, and the number of LOC modified tend to increase as we move from PCE\textsubscript{simple} to PCE\textsubscript{unified}. Functionality of all six modules needs to be tested in PCE\textsubscript{unified}, although the change affects only one module. This could be a major burden, given the immaturity of WA testing techniques. In general, clone unification limits the degree of freedom with which individual clones can evolve independently of the others. Therefore, while clone unification may ease certain kind of modifications (typically, modifications that needs to be repeated for multiple cones, such as given in Table 2) it can also render certain other kind of changes more difficult to do (typically, localized modifications applicable to a minority of the clones, such as given in Table 3).

4.8 Efficiency of source code packaging

Often, the Server page portion of a WA is delivered in source form. In such cases it is desirable to eliminate all the unused code from the delivered code. This may be due to space/time efficiency concerns (e.g., severe space constrains on the server) or to avoid transfer of unused client-side scripts over the network. Or this may be to minimize impact of modifications. Most WAs are accessed globally, and need to be available 24/7. Downtime caused by updates to the unused code is unacceptable for such WAs. Unfortunately, clone unification sometimes injects unused code into the delivered code. For example, in PCE\textsubscript{unified} Staff module uses only 77% of the unified module. If the unified module is reused in another WA to serve as a Staff module, it results in carrying over 23% of the code that will not be used at all. Therefore clone unification can sometimes inject unused code into the distribution package, forcing a trade-off in our ability to distribute a clean, minimal, source code package.

4.9 Ability to vary runtime structure

Occasionally it may be necessary to have a different runtime structure between cloned systems. Possible reasons for this include:

- to fit a new API/framework/platform (e.g., to deploy PCE modules on a different Foundation)
- when one clone require better performance than the rest (e.g., PCE\textsubscript{simple} Vs PCE\textsubscript{unified} )
- for compatibility with other legacy systems at the deployment-site (e.g., to integrate with a legacy system that uses an old version of PHP)

Although our reasons for having three PCEs were quite different from those given above, we too found ourselves in a similar scenario: We had to maintain three separate WAs having drastically different...
runtime structures, yet having much similarity among them. For example, 55% of the code of PCE\textsubscript{simple} was found to have a cloned counterpart in PCE\textsubscript{patterns}. Unification of such clones requires some re-alignment in the runtime structures, forcing trade-offs in the motives behind varying the runtime structures in the first place.

5. PCE on J2EE and .NET

J2EE\textsuperscript{TM} and .NET\textsuperscript{TM} are the two advanced platforms for implementing WAs. They provide rich sets of general middleware-level infrastructure services (e.g., for managing security, transactions, resources). In our PHP solution, the Foundation provides similar service, but in addition, it also provides more application-specific infrastructure services. Therefore, PCE implemented on the .NET or J2EE is likely to follow the same high-level architecture shown in Figure 5, possibly with a thinner Foundation (since some middleware-level services are provided by the platform itself). As the design patterns we applied in our experiment are also applicable to .NET and J2EE, we expect to see similar cloning situations across PHP, .NET and J2EE platforms. Also, the basic role of Server pages remains the same whether we use PHP, ASP.NET or JSP on J2EE. Therefore, we believe that the limits involved in using Server pages, at least in the context of situations discussed in Section 4, apply independently of the platform on which these techniques are used. But further work is required to support or dismiss this claim. One area such further work should consider is the additional UI technologies offered by .NET and J2EE. Such UI technologies in J2EE include 


custom tags, Expression Language all help in implementing the Template view pattern and Composite View pattern. Similarly, .NETs’ Code-Behind feature provides clean separation of HTML from generation logic, while Server Controls aid in rapid constructing of WA UIs. ASP.NET also has built in support for Page controller pattern. ASP.NET Webparts and Java Portlet API provide building blocks for content aggregation in Portal type WAs. Other related technologies requiring similar further work include web application frameworks (e.g., Struts, Ruby on Rails), incarnations of Server page technique in other languages (e.g., ColdFusion), template engines (e.g., Velocity), and transformation techniques (e.g., XSLT).

6. Related work

Cloning is a well known problem in traditional software development, and it has been under research for more than a decade. Recently, cloning in web domain has started to attract interest from the research community. Some work on detecting clones in web domain (e.g., [2][6][7][8][14][22]), and reengineering clones in legacy WAs have been reported (e.g., [2][7][22]). To the best of our knowledge, there has been no previous in-depth treatment of trade-offs involved in clone unification.

Cordy’s work [3] in critical financial systems reports that the risk of breaking an existing system is a great deterrent to clone unification. We can formulate this as a trade-off, i.e., clone unification forces a trade-off in system reliability. He also mentions that certain clones speed up development/maintenance by introducing a ‘degree of freedom’. Although not the main focus of their work, Synytskyy et al. [22] point out that overzealous clone unification can result in hard-to-understand spaghetti code. We agree with both these views, as implied by Sections 4.6 and 4.7. Synytskyy et al. also mention how multiple content types complicate clone detection in web domain. We observed that the same is true for clone unification (cf Section 4.5).

Bolodyreff and Kewish [2] propose to store unified clones in a relational database, and to retrieve the clone at runtime using scripts. A somewhat similar approach used by Ricca and Tonella [21]. Clone unification by storing cloned web page fragments in a database in this manner is a powerful mechanism with its own merits. However, it should be used in moderation as it may aggravate trade-offs in a number of areas, such as in performance, WYSIWYG editing, and indexing by search engines.

Clone unification proposed by [2][21] and [22] is automatic ([21] allows manual refinements to the generated result). Work by Ping and Kontogiannis [19] proposes an approach to automatically refactor web sites that removes some ‘potential duplication’. Such automation is a step forward as it greatly reduces the effort required in clone unification. However, one needs to be careful not to set off the advantages of automation with the cost of tradeoffs we have highlighted here.

7. Conclusions and future work

This work is a follow up on our earlier discovery of high levels of cloning in WAs [20]. Using an empirical study, we showed that it was technically feasible to use Sever pages to unify most of the clones. Such unification greatly reduced the code size and the chance of update anomalies. However, this approach forced trade-offs in many important WA properties. In a real-world WA project, these trade-offs limit how far we can practically push Server pages towards clone unification.

Our findings in this paper indicate that we may have to incur many trade-offs when tackling clones,
particularly those in WAs. But this should not be interpreted as an argument against clone unification. Rather, we believe that it sheds more light as to why clones persist in software, in spite of many techniques to avoid them. By doing so, it provides us with a critical success criterion against which solutions to cloning should be evaluated against. That is, such a solution has to take into account possible trade-offs it may incur in other important properties of the software, such as those we observed in our experiment.

The long-term goal of our research is to find effective methods to combat cloning. One promising direction we are pursuing is the use of generative programming to tackle clones. Our approach does not unify clones in the program code, but it does so at the meta-level program representation, from which an executable program can be automatically obtained. This is particularly suitable in situations such as described in this paper, when removing clones triggers undesirable impact on other software qualities that cannot be compromised. It also applies in situations when clones in the program serve some useful purpose (e.g., to improve performance, or to conform to platform requirements). The maintenance of a program is done at the non-redundant meta-program level. Industrial application [16][26] and lab studies [25] indicate that such an approach may bring considerable productivity gains. We plan to investigate how this approach can fare in terms of avoiding the trade-offs we observed in this experiment.

8. References

[10] Fowler M., Refactoring - improving the design of existing code, Addison-Wesley, 1999
[26] Zhang, W. and Jarzabek, S. “Reuse without Compromising Performance: Experience from RPG Software