Build from the End

PHILIPP BACHMANN, iRIX Software Engineering AG

This pattern describes how to build products, e.g. how to make a software application using Make or Ant, in correct order just by explicitly stating both dependencies of artifacts from their ingredients and rules to produce each single artifact. Delegating the ordering of the building steps to a tool helps remaining flexible and allows for automated parallelization of execution. This is done by separating the whole build task into three steps: Specification of each single artifact, its direct prerequisites and a rule that tells how to make the artifact from its prerequisites, using a tool to process these specifications into a global graph and finally executing the graph.
This pattern is primarily aimed at developers starting to implement and build an application partitioned in several source files. The same pattern also can aid project managers in scheduling.
Both a real world example and sample code accompany the pattern presentation. The sample code is a Makefile.
The presentation of the pattern follows the style well known from [Buschmann et al. 2000b] and [Schmidt et al. 2002]. This pattern is based upon other patterns. Typographic conventions for references to other patterns are similar to [Alexander et al. 1977]. A Glossary provides thumbnails of many of these patterns and definitions of other terms used in this paper.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.9 [**Software Engineering**]: Management—*Software process models*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.3.4 [**Programming Languages**]: Processors—*Interpreters*; F.2.2 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Scheduling*

General Terms: Design

Additional Key Words and Phrases: Patterns, Processes, Depencencies, Correctness

ACM Reference Format:

Bachmann, Ph. 2013. Build from the End, HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 20 (October 2013), 12 pages.

1. INTENT

This pattern describes how to build products, e.g. how to make a software application using Make or Ant, in correct order just by explicitly stating both dependencies of artifacts from their ingredients and rules to produce each single artifact. Delegating the ordering of the building steps to a tool helps remaining flexible and allows for automated parallelization of execution.

2. EXAMPLE

Imagine you are developing a software application. The source code and binary libraries are distributed among several files. The final application depends on all of them, more specific source files defining e.g. derived classes depend on more general source files in Java, which contain e.g. declarations of interfaces or abstract base classes, or in C and C++ on so-called header files, which contain class declarations. Building the application

This work is supported by iRIX Software Engineering AG.
Author's address: Ph. Bachmann, c/o iRIX Software Engineering AG, Dornacher Str. 192, 4053 Basel, BS, Switzerland; email: bachlipp@web.de

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 20th Conference on Pattern Languages of Programs (PLoP). PLoP'13, October 23–26, Monticello, Illinois, USA. Copyright 2013 is held by the author(s). Parts of the Glossary first published in [Bachmann 2008]: Copyright 2008–2013 by the author and the Institute for Medical Informatics and Biostatistics, used here with permission from F. Hoffmann-La Roche Ltd. HILLSIDE 978-1-941652-00-8

requires several intermediate steps, e.g. compiling source files into object files and link editing all resulting object files into the binary representing the application. To do so, a certain order has to be followed resulting in a build process. This order is determined by the dependencies among the files and artifacts that go into the build. Figure 1 shows basic header and source files on the left, intermediate object files gained by compiling the source files in the middle and the binary finally resulting from link editing the object files on the right.

Most often only a small fraction of all files the application will be built from gets changed at a time. Therefore it is equally important to note which files do *not* depend on each other. Otherwise the whole build process had to be repeated all over again whenever an arbitrary file was changed. Taking into account which files do not depend on each other allows for partially rebuilding the application, which usually results in huge performance gains facing the fact that changes are in general small with respect to the whole application; if `derived1.cxx` had to be changed for example, then it suffices to recompile it yielding `derived1.o` and link editing all object files again resulting in a new version of the binary `result`. Furthermore, the information which files are independent from each other might also be used to parallelize the build process, e.g. taking advantage of multi-core processors; in the example above, all three object files could have been compiled in parallel from their respective source files.

During application development the build process will be repeatedly executed, probably by more than one person. Therefore it is obvious that it will pay off to reify the build process itself in software. Along with the evolution of the application its partitioning in several files will evolve, so its build system will also evolve. Therefore adaptability is an important consideration when designing and using build systems.

3. CONTEXT

Building non-trivial products requires a series of tasks to be applied in correct order. Examples of such products include software applications and the results expected from running a project. The sequence can be formalized in terms of a build process. For each final product the process is going to be enacted once. Each single task transforms one or more inputs into most often one single output and adds value in doing so. At the core of applying the tasks in correct order are dependencies of artifacts from more basic artifacts. This is meant both causally and temporally: A higher-level artifact can only be assembled from lower-level ingredients, if and only if these ingredients are already there.

This applies recursively.

The dependency relationships must not contain any cycles—an artifact must not directly or indirectly depend on itself. Sometimes this requirement can only be fulfilled when considering coarser-grained sets of the original artifacts instead of these artifacts themselves.

Artifacts undergo maintenance and evolution. So the dependencies relationships may also evolve.

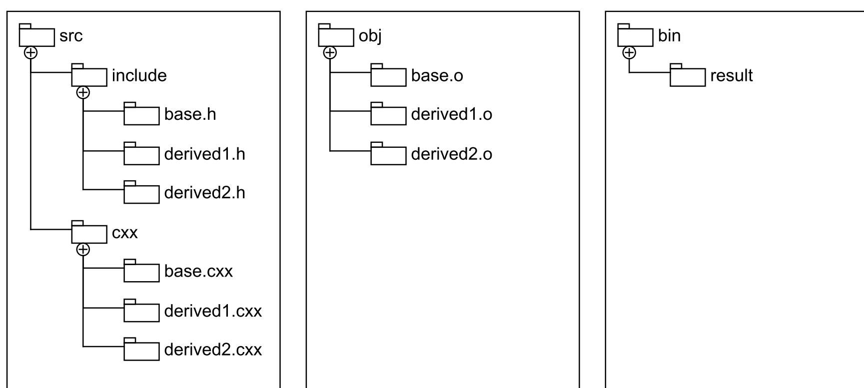


Fig. 1: Source files and their transformation into an executable artifact

4. PROBLEM

How to define the build process such that it proves reliable, efficient, adaptable, and parallelizable and can be performed automatically on request?

5. FORCES

—An explicit representation of the process itself is hard to maintain.

—Software builds have to be correct. Updates to single files should imply rebuilding them and to rebuild all artifacts that depend on them—so the relative ordering must be respected.

—Software builds should be as fast as possible. Only those files should be recompiled and link edited that really have to.

—Build systems are necessary for build automation; build automation enables continuous integration.

—Build systems are software themselves, so they have to be treated as such, e.g. they undergo changes, will profit from support of configuration management systems, need documentation, testing etc.

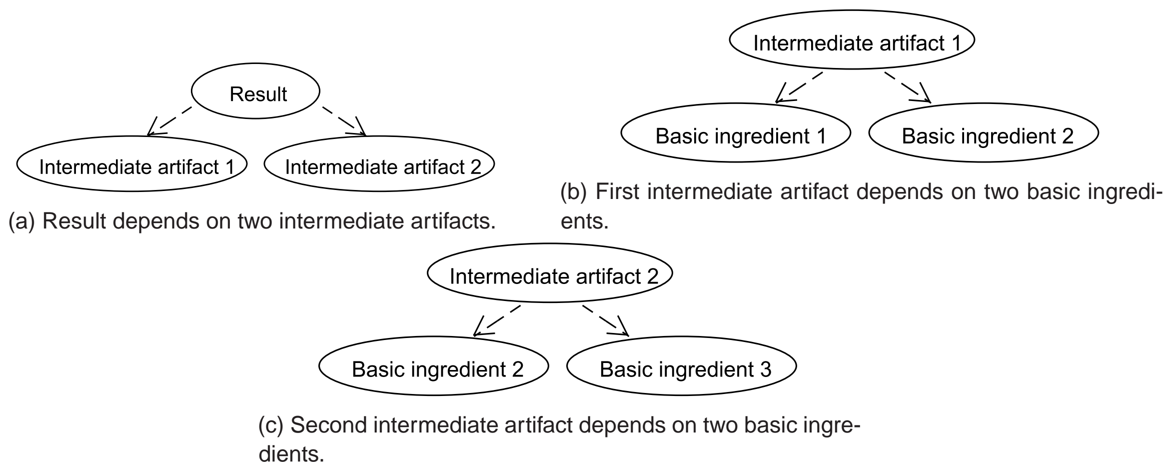
—Modern hardware provides processors with multiple cores—why not use these?

6. SOLUTION

Think from the end, what your result should be. Perceive the goal as the result of prerequisites, which in turn depend on other prerequisites. So work your way backwards until you have recursively covered all the dependencies for the build.

Instead of explicitly setting up a sequence of build steps, specify only the dependencies of artifacts and the rules on how to transform one or more artifacts into a higher-level artifact. Let a tool then figure out the correct sequence and build the stuff.

So you will recursively end up with a set of both declarations that define artifacts and their dependencies and the respective atomic instructions on how to transform one or more prerequisites into (intermediate or final) artifacts. Figure 2 graphically depicts such dependency declarations; the arrows read “depends on”: (a) graphically depicts that the result depends on two intermediate artifacts, (b) says, that furthermore the first intermediate artifact depends on two basic ingredients, and (c) finally tells us, that the second intermediate artifact depends on two basic ingredients. Note that one of the three basic ingredients is required by both intermediate artifacts.



(a) Result depends on two intermediate artifacts.

(b) First intermediate artifact depends on two basic ingredients.

(c) Second intermediate artifact depends on two basic ingredients.

Fig. 2: A complete set of dependency declarations

Given there are no cycles as required in Section 3, the dependency relation establishes a Strict Partial Order.

For such kinds of orders several topological sorting algorithms exist to connect the set into a directed acyclic graph (see Section 6.5.3 for a reference). The graph for Figure 2 is shown in Figure 3. This graph reifies the process from start to end. It tells you which artifacts do not depend on other artifacts—these are the starting points—and shows possibilities to do things concurrently.

6.1 Participants

Table I shows Candidate–Responsibility–Collaboration Cards for the aforementioned participants. Each single participant is commented on further below:

Artifact. Final and intermediate artifacts that have to be built from other Artifacts and axiomatic artifacts that already exist.

Client. Supplies Artifact dependencies and Transformer rules to TopologicalSorter and Director and finally get built Artifacts in return.

DirectedAcyclicGraph. Represents total order of artifacts, thus allows for enactment to build result from ingredients in correct order.

Director. Orchestrates the enactment of the DirectedAcyclicGraph concurrently, taking availability of processing resources into account. This is an example of the Director role from the BUILDER [Gamma et al. 1996c] design pattern.

TopologicalSorter. Establishes directed acylic graph from dependency declarations.

Transformer. Builds higher-level Artifacts from more basic Artifacts according to some given rule.

6.2 Dynamics

Instead of planning from the beginning to the end, i.e. establishing an explicit flow, only specify the dependencies. Do so starting from the end. Then let a tool generate DirectedAcyclicGraph. Enacting the graph afterwards lets the tool finally build your product. Building and enacting the graph are often combined in practice.

The dynamics of this pattern is shown in Figure 4.

6.3 Rationale

Building non-trivial products can get quite complex, because the products to be built are complex. The real value lies in the products, however, so the goal is for the main effort to go into developing the products themselves, not into perfecting the respective build processes. A build process that refuses adaptation because of its inherent complexity will also slow down product development. The solution for the challenge of building products proposes to specify only the minimum—and let a tool do the rest.

Coupling production steps only by means of their dependencies is a form of loose coupling. Because of loose coupling, the whole build process can be easily adapted and parallelized. Both aspects are elaborated on below.

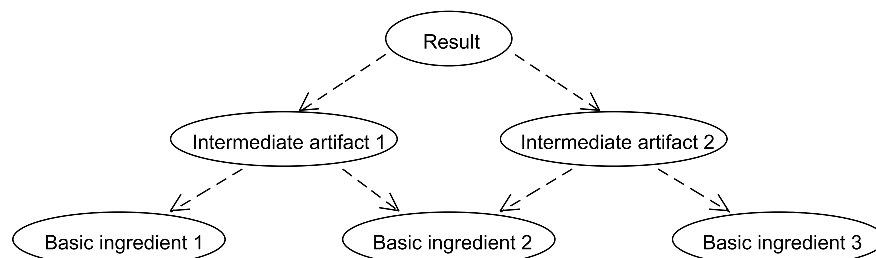


Fig. 3: Directed acyclic graph corresponding to Figure 2

Table I. : Candidate–Responsibility–Collaboration Cards

(a) Artifact

Artifact	
carries	date of last modification
either atomic or depends on	Artifacts

(b) Client

Client	
Contracts	TopologicalSorter
Contracts	Director

(c) DirectedAcyclicGraph

DirectedAcyclicGraph	
Connects in order of dependencies	all Artifacts

(d) Director

Director	
Enacts	DirectedAcyclicGraph
to build	all Artifacts
delegating work to	Transformer

(e) TopologicalSorter

TopologicalSorter	
Establishes	DirectedAcyclicGraph

(f) Transformer

Transformer	
Builds	Artifact
from	Artifacts it directly depends on

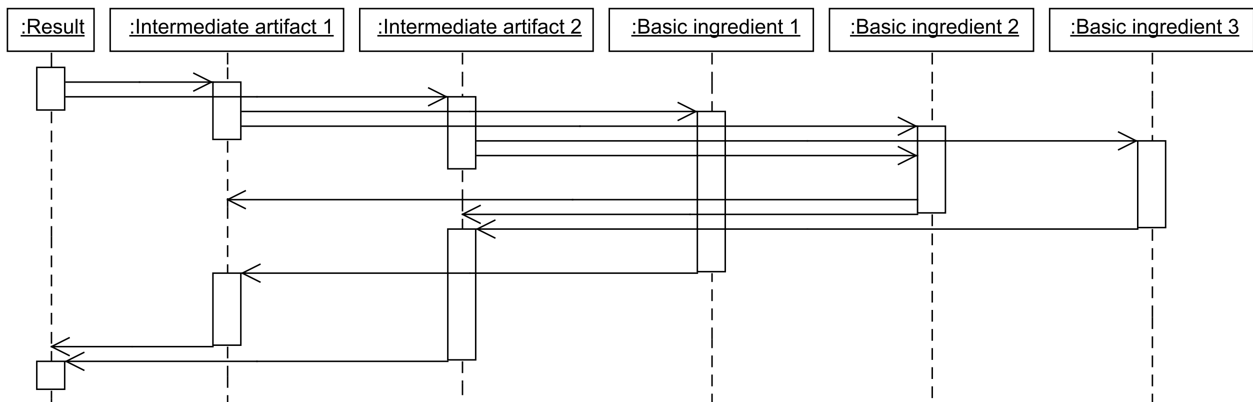
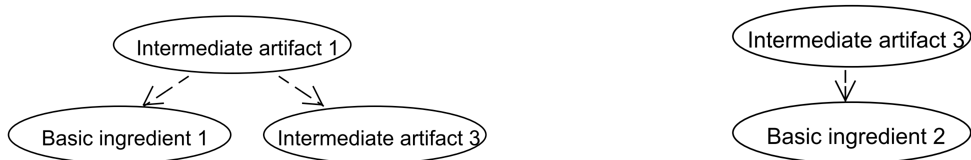


Fig. 4: Sequence diagram illustrating Build from the End

This contributes to flexibility: Because the graph can always be generated again, it is an easy task to modify dependency relations and thus modify overall production. Consider for example the effort required introducing another intermediate artifact, thus e.g. substituting new dependencies in Figure 5 for the dependency in Figure 2b. Again note that both Figures 2c and 5b refer to the same basic ingredient. Figure 6 shows the respective DirectedAcyclicGraph, Figure 7 the corresponding sequence diagram.

Loose coupling provides an important side effect: For the machine it is quite easy to detect opportunities for parallelization, because the dependencies are explicitly given and no complicated dependency and aliasing analyses have to be performed first as e.g. in automatic loop parallelization. At any single state of enactment of the graph any nodes with dependencies that have all been satisfied can be processed in parallel.

Process diagrams can get remarkably complex. Even though diagrams could still be created from the generated directed acyclic graphs, it is not really necessary because process control can also be automated. So the effort otherwise necessary to comprehend the process flow can now be utilized to achieve the overall goal.



(a) First intermediate artifact depends on another intermediate one and a basic ingredient.

(b) Third intermediate artifact depends on a basic ingredient.

Fig. 5: Evolution of Figure 2b

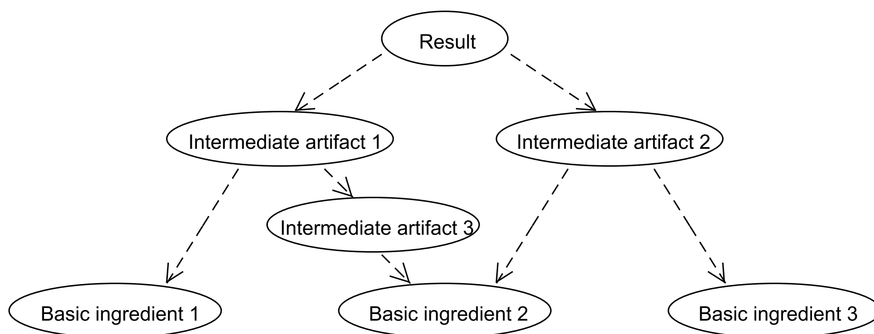


Fig. 6: Directed acyclic graph corresponding to Figure 5

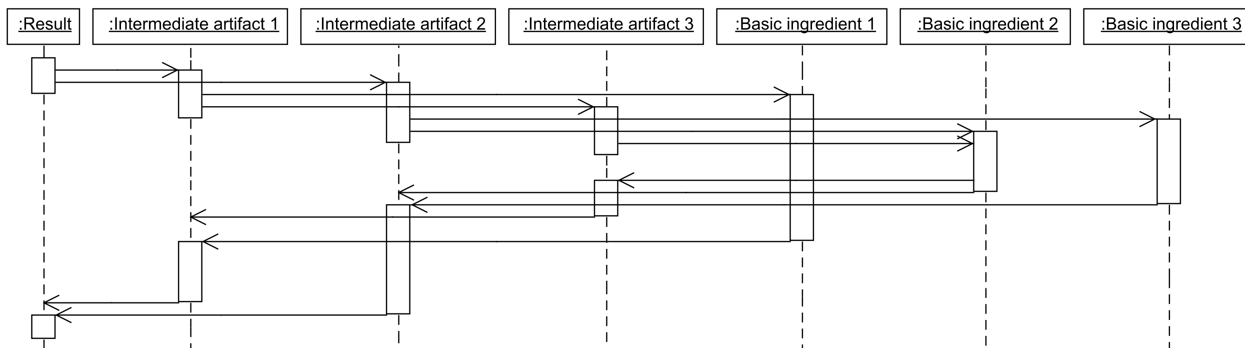


Fig. 7: Sequence diagram illustrating modified process

6.4 Resulting Context

Now we end up with three entities:

(1) A list of dependency and rule specifications,

(2) a TopologicalSorter that generates the global DirectedAcylicGraph, and

(3) a Director that follows the graph and calls Transformer to build artifacts represented by the nodes in the graph from more basic artifacts to build the overall product.

Steps 1 and 2 also form the meta-process of building the build system. Step 3 forms the process of building the product itself.

6.4.1 *Consequences.* The BUILD FROM THE END pattern has the following benefits:

(1) *Adaptability.* The process execution as a whole is not tangible any more. In fact, it depends on the dependency specifications, the TopologicalSorter and the Director, so it may “constantly” change.
(2) *Dependency specifications are lightweight.* You do not rely on an explicitly defined, heavyweight process.
(3) *Textual language.* Many topological sorters process text files as input. Configuration management systems in general provide powerful tools to e.g. compare different versions of such files.
(4) *Concurrency can be gotten “for free”.* Without any additional effort on the Client side the Director can automatically employ parallelism.

The BUILD FROM THE END pattern has the following liabilities:

(1) *Detail view.* The Client only specifies the relative dependencies. At least without tool support for visualization of DirectedAcyclicGraph, you may wish to have a better bird’s eye view of the whole.
(2) *More complex.* You rely on additional tools, the TopologicalSorter and the Director.
(3) *Intermediate artifacts may become orphans.* Not all tools around keep track of whether intermediate artifacts are still necessary, so they cannot safely remove them again from the filesystem. In this case at some point a large fraction of the whole software project will consist of superfluous files.
(4) *Nondeterministic order of execution.* If not all dependency relations present have been diligently fed to the TopologicalSorter, then the Director might update less artifacts than necessary upon changing a prerequisite. You cannot rely on any additional order of execution but those given by the dependencies stated. This is especially true for parallelizing Directors. Therefore you have to completely specify the dependencies.

6.5 Implementation

In most cases there already are implementations of TopologicalSorter and Director and for the specific domain, so implementation of this pattern primarily consists of specifying the artifacts, their prerequisites and the Transformer rules in a language the TopologicalSorter can understand. Section 6.5.1 contains a complete implementation for this simpler case.

It gets more complicated if there are no existing implementations of TopologicalSorter or the Director. This case is only covered briefly in Section 6.5.3.

6.5.1 *Example Resolved.* For building software applications several tools are available. In this example GNU Make is going to be used to build an application written in the C++ programming language. This application has already been sketched out in Section 2; for the sake of readability the folder structure proposed there is flattened out here. Make is an example for both a domain-specific language to specify software builds by a combination of dependency declarations and commands to build artifacts from ingredients and a combination of TopologicalSorter and Director that builds the final artifact based on Makefiles written in this language. Section 6.6 points to some similar tools available.

The details of the source code do not matter, it suffices to say that in file `base.h` a base class is declared, which is defined in file `base.cxx`; files `derived1.h` and `derived2.h` include `base.h`, because in both files classes specializing the base class are being declared, in turn defined in the corresponding `.cxx` files. The dependency structure is being fed to Make in a file called a Makefile. Each line that starts in the first column lists an artifact to be built—the target in Make terms—, a colon and a space separated list of so-called prerequisites considered necessary to build the target. The instructions on how to transform the prerequisites into the target follow on lines each indented with a tabulator (!). With Make, targets and prerequisites are not just names—by default Make treats them as files, which may or may not exist and which may differ in their date of last modification. Make will build or update a target if and only if it does not exist yet or at least one of its prerequisites is more recent than the target.

The complete picture is given in Listing 1.

Listing 1: Makefile

```
SHELL = /bin/sh

.SUFFIXES:

.INTERMEDIATE: derived1.o derived2.o base.o

.PHONY: all

all: result

result: derived1.o derived2.o base.o
	g++ $< -o $@

derived1.o: derived1.cxx
	g++ -c $< -o $@

derived2.o: derived2.cxx
	g++ -c $< -o $@

base.o: base.cxx
	g++ -c $< -o $@

derived1.cxx: derived1.h

derived2.cxx: derived2.h

base.cxx: base.h

derived1.h: base.h

derived2.h: base.h

base.h: Makefile
```

Note some details of this implementation: The special target `.SUFFIXES` switches off built in suffix rules to allow for a more explicit control of what is going on. `.INTERMEDIATE` marks its prerequisites as intermediate files, i.e. if Make has built them, it will remove them again right before terminating. `.PHONY` says that `all` is just a name and not a file—`all` is a special name in Make, because it is the default target to run if no alternative targets are given on the command line. The next line is just for convenience and states that `result` has to be built before we can say to have succeeded in building `all`. Given the Makefile has been saved under one of the names considered default by Make, e.g. `Makefile`, it therefore suffices to just type `make` from a command line shell. The remainder of the Makefile is specific to the example application. The `result:` lines tell Make how to link edit `result` from three object files; `$<` is a shorthand for all respective direct prerequisites, while `$@` always denotes the respective target. Next follow three blocks that tell Make how to compile `.cxx` files into object files. The rest of the Makefile shown is just another representation of the preprocessor directives in the source and header files to include header files—these lines could also have been dynamically generated with help of the C++ compiler—in the case of the GNU Compiler Collection using the command line options `-M -MG`—and including the respective compiler output into the Makefile, see [Miller 1997] for a description of this technique. Note that the Makefile imlicitly got declared as a prerequisite for anything else to ensure full rebuilds to take place after editing the Makefile, e.g. changing compilation options.

This Makefile shows several occurrences for plain dependency specifications without Transformer rules. In all of those cases the target either has been declared `.PHONY` or has to be provided by the Client as it is not meant to be generated by the build process; in the latter case the dependency declarations serve just to pass the most recent date of last modification of the prerequisites on to the respective target, if and only if this target itself was last modified before that date. Doing so ensures that no updates of indirect prerequisites get missed by the build process.

6.5.2 *Relationship of Example and Participants.* The software building example laid out in Sections 2 and 6.5.1 map to the participants defined in Section 6.1 as shown in Table II.

6.5.3 *Implementing Build Tools from Scratch*

> The Future is Parallel, and the Future of Parallel is Declarative.
>
> SIMON PEYTON JONES

One of the simplest implementations of a combination of TopologicalSorter and Director uses Kahn's algorithm. The result of of sorting is a linear list of build steps to perform that respects the partial order given by the dependency relations. So when iterating through such a list it is always guaranteed that all prerequisites have already been visited before visiting the artifact that needs all of them. Kahn's algorithm does not explicitly represent the DirectedAcyclicGraph, however. Therefore the Director cannot execute independent tasks concurrently. For parallelization of Directors, more sophisticated TopologicalSorters are therefore necessary.

6.6 Known Uses

Examples of this pattern can be found in existing software.

6.6.1 *Ant.* Apache Ant is well-known in the Java community. Its buildfiles conform to an XML schema. The actions to perform are called tasks, and they are portable, because they have been written in Java. With help of the Java classloader even tasks not part of the standard Ant distribution can be plugged in at runtime.

In the Java ecosystem the Java compiler itself already handles certain subtasks typical for build tools. Given Ant builds a Java application, Ant itself therefore only has to handle higher-level tasks such as triggering compilation as a whole and assembling JAR files.

6.6.2 *Jam.* Originally developed by Perforce Software, its variant BJam is the core of Boost.Build, the build system of the Boost C++ libraries. Both Jam and BJam qualify as open-source software. With a build system implemented on top of Jam or its derivatives dependencies of more than one target from multiple prerequisites can be concisely expressed. Furthermore, the commands to actually transform prerequisites into targets can be factored out into so-called actions.

Table II. : Relationship of Examples and Participants

Code	Participant
`base.h`, `base.cxx`, `base.o`, `derived1.h`, `derived1.cxx`, `derived1.o`, `derived2.h`, `derived2.cxx`, `derived2.o`, `result`	Artifact
The person who has written the Makefile and calls Make	Client
Implicit. Can be made explicit when calling Make e.g. with its `-d` option.	DirectedAcyclicGraph
Make	TopologicalSorter, Director
`g++ $< -o $@`, `g++ -c $< -o $@`	Transformer

6.6.3 *Make.* Make has been used as the guiding example thoughout this paper. The actions to perform are called commands, and these are going to be executed by the shell, so they are specific to the platform Make runs on—thus Makefiles are not portable. There are several implementations of Make which differ in the exact syntax of Makefiles. Some like GNU Make allow for very brief Makefiles due to built-in suffix and pattern rules that help reduce repetitive expressions in Makefiles. [Stallman and McGrath 1998; Miller 1997] contain a lot of useful tips. There is a Perl project called Makefile-GraphViz that visualizes Makefiles.

6.6.4 *Tup.* Tup—an acronym for "the updater"—is a build system that stores the dependency relations in an embedded database, here SQLite. Tup provides advanced capabilities to visualize the directed acylic graph. The algorithms have been documented in [Shal 2009].

6.6.5 *Project Management Methods and Tools.* This pattern is being used even outside the domain of software development, and has been used for a very long time. Examples include the Critical Path method and the Program Evaluation and Review Technique (PERT). The visual representation of the directed acyclic graph is often called the project network within this context. Software that assists in project management like Microsoft Project and TaskJuggler can transform activities and their dependencies into the project network while also respecting resource constraints. Employing advanced and flexible project planning methods is a step towards organizational maturity higher than CMM(I) 2 ("Repeatable / Managed") and beyond Software Subcultural Pattern 2 ("Routine") [Weinberg 1992, pp 24–35,103–104].

Project management takes a more coarse-grained view than building products, however. Otherwise project managers would not see the wood for the trees. Typical categories are for example requirements engineering and architecture. These are that coarse that they mutually depend on each other and are executed in parallel—architects wait for requirements engineers, and those need the input from architects in turn. This is especially true in early, more elaborative phases of a project, e.g. in Unified Process in the phase called Elaboration.

6.7 Related Computational Models and Patterns

This pattern is related to Dependency Network, one of the alternative computational models described in [Fowler 2011]. While FOWLER puts emphasis on certain aspects of the domain-specific language the artifacts and prerequisites have to be described in and makes a difference between product-oriented (e.g. Make) and task-oriented (e.g. Ant) styles of Dependency Networks, the core of this pattern says that using the solution proposed can let you focus on the final result you are trying to accomplish; furthermore the opportunity of automatic parallization is being stressed here.

The TopologicalSorter is an example of INTERPRETER [Gamma et al. 1996d], that builds DirectedAcyclicGraph from the dependency specifications given. The Director is a role taken from the BUILDER [Gamma et al. 1996c] design pattern and is an INTERPRETER, too, that builds COMMANDS [Gamma et al. 1996a] ordered by DirectedAcyclicGraph and feds them into a COMMAND PROCESSOR [Buschmann et al. 2000a].

6.8 Summary

If your perspective is from the end goal, it will be less likely that you get bogged down in details. BUILD FROM THE END gives you both a tool to do so and lets you concentrate on developing the product instead of perfecting the build tool, because the input to tools like Make and Ant is quite lightweight—the real work is then done by the tool.

APPENDIX

REFERENCES

Ademar Aguiar and Joseph Yoder (Eds.). 2008. *PLoP '08: Proceedings of the 15th Conference on Pattern Languages of Programs.* ACM, New York, NY, USA.

Christopher Alexander, Sara Ishikawa, and Murray Silverstein. 1977. *A Pattern Language. Towns, Buildings, Construction. With* MAX JACOBSON, INGRID FIKSDAHL-KING *and* SHLOMO ANGEL. Oxford University Press, New York.

Philipp Bachmann. 2008. Deferred cancellation. A behavioral pattern, see Aguiar and Yoder [2008], 1–17. DOI:http://dx.doi.org/10.1145/1753196.1753218

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 2000a. *Command-Processor* (1998, 1. korr. Nachdruck), Chapter 3: Entwurfsmuster, 277–291. In Buschmann et al. [2000b]. German translation of "Command Processor".

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 2000b. *Pattern-orientierte Softwarearchitektur. Ein Pattern-System, deutsche Übersetzung von* CHRISTIANE LÖCKENHOFF (1998, 1. korr. Nachdruck). Addison-Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam. German translation of "Pattern-Oriented Software Architecture. A System of Patterns".

Martin Fowler. 2011. *Domain Specific Languages. With* REBECCA PARSONS. Addison-Wesley, Upper Saddle River, NJ · Boston · Indianapolis · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Sydney · Tokyo · Singapore · Mexico City.

Richard P. Gabriel. 2002. *Writers' Workshops & the Work of Making Things. Patterns, Poetry. . . .* Addison-Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1996a. *Befehl* (dritter, unveränderter Nachdruck), Chapter 5: Verhaltensmuster, 273–286. In *Professionelle Softwareentwicklung* Gamma et al. [1996b]. German translation of "Command".

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1996b. *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, deutsche Übersetzung von* DIRK RIEHLE (dritter, unveränderter Nachdruck). Addison-Wesley-Longman, Bonn · Reading, Massachusetts · Menlo Park, California · New York · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam. German translation of "Design Patterns. Elements of Reusable Object-Oriented Software".

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1996c. *Erbauer* (dritter, unveränderter Nachdruck), Chapter 3: Erzeugungsmuster, 119–130. In *Professionelle Softwareentwicklung* Gamma et al. [1996b]. German translation of "Builder".

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1996d. *Interpreter* (dritter, unveränderter Nachdruck), Chapter 5: Verhaltensmuster, 319–334. In *Professionelle Softwareentwicklung* Gamma et al. [1996b]. German translation of "Interpreter".

Peter Miller. 1997. Recursive Make Considered Harmful. Published online (Aug. 1997). Retrieved January 9, 2007 from http://www.canb.auug.org.au/ millerp/rmch/recu-make-cons-harm.html millerp@canb.auug.org.au.

Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. 2002. *Pattern-orientierte Software-Architektur. Muster für nebenläufige und vernetzte Objekte, übersetzt aus dem Amerikanischen von* MARTINA BUSCHMANN. dpunkt.verlag, Heidelberg. German translation of "Pattern-Oriented Software Architecture. Volume 2: Patterns for Concurrent and Networked Objects".

Mike Shal. 2009. Build System Rules and Algorithms. Published online (2009). Retrieved July 18, 2013 from http://gittup.org/tup/build_system_rules_and_algorithms.pdf marfey@gmail.com.

Richard M. Stallman and Roland McGrath. 1998. *GNU Make. A Program for Directing Recompilation, GNU make Version 3.77.* Free Software Foundation, Boston, Massachusetts.

Gerald M. Weinberg. 1992. *Quality Software Management.* Vol. 1: Systems Thinking. Dorset House Publishing, New York, New York.

Glossary

This section contains thumbnails and definitions of the most important patterns and terms used in this paper.

Builder. The BUILDER pattern separates the creation of a complex object from its representation. BUILDERS allow for creation of a variety of representations, and the set of representations the builder can build can evolve over time without modifying the representations.[Gamma et al. 1996c]

Command. The COMMAND pattern encapsulates commands as objects. COMMANDs can be placed in a queue and are an essential ingredient for undo functionality.[Gamma et al. 1996a]

Command Processor. A COMMAND PROCESSOR manages the execution of COMMANDs. It mediates between the invocation and the actual execution of a COMMAND and might provide further services to e.g. undo a COMMAND again.[Buschmann et al. 2000a]

Interpreter. If problems of a certain kind repeatedly occur, then it might pay off to define a formal language problems of this kind can be expressed in and build an interpreter that can interpret problems represented in this language.[Gamma et al. 1996d]

Portability. Portability has two aspects: Portability with regard to environments or platforms means that an application requires no or only a few local changes to run on another platform than once planned for. The term platform can refer to operating system, hardware architecture or even a set of third party software the application interfaces with, e.g. a database management system. Portability in time means that an application can still be compiled after years have passed.

ACKNOWLEDGMENTS

Without the invaluable feedback of Joseph W. Yoder, who was the PLoP shepherd of this work, this paper would not have been the way it is now.

The author would like to thank all the participants of PLoP 2013 for their contributions. Special thanks go to the members of the Writers' Workshop [Gabriel 2002] "000101" the author participated in: Paul M. Chalekian, Alfredo Goldman vel Lejbman, Lise B. Hvatum, David A. Mundie, Alexander Nowak, William F. Opdyke and last but not least Andreas Fießer and Lori Flynn, its moderators. Their feedback had a significant impact on the current version of the paper.

Last but not least my thanks and love go to Cornelia Kneser, my wife, for her constant support throughout the writing of the paper.

Received May 2013; revised October 2013; accepted May 2014

PLoP'13, October 23–26, Monticello, Illinois, USA. Copyright 2013 is held by the author(s). Parts of the Glossary first published in [Bachmann 2008]: Copyright 2008–2013 by the author and the Institute for Medical Informatics and Biostatistics, used here with permission from F. Hoffmann-La Roche Ltd. HILLSIDE 978-1-941652-00-8