Architectural Smell Definitions and Formalizations

Joshua Garcia
March 16, 2014

1 Architectural Concept Formalizations for Smell-Relevant View

In this section, we provide definitions of basic software architectural concepts and use them to
define architectural smells. Our definitions are not intended to be complete; they are restricted to
those architectural concepts that will be useful for identifying smells. We also provide shorthand
predicates in Figure [1] that we use to help us define architectural smells.

A software system’s architecture is a graph $G$ whose vertices are “bricks” (software components
and connectors) and whose topology represents the interconnections among those bricks. In order
to represent and detect architectural smells, we model a system’s architecture as a tuple comprising
$G$, the nonempty set of “words” $W$ that are used to “describe” (i.e., implement) the system modeled
by the architecture, and the nonempty set of “topics” $T$ addressed by the system; each topic is
defined as a probability distribution over the system’s words. By examining the words that have
the highest probabilities in a topic, the meaning of that topic can be discerned. In this way, a topic
can serve as a representation of a concern addressed by a software system. In other words, the set
of topics $T$ is a representation of the system’s concerns.

$$A = (G, W, T)$$
$$G = (B, L)$$
$$W = \{w_i \mid i \in \mathbb{N}\}$$
$$T = \{z_i \mid i \in \mathbb{N}\}$$
$$z = Pd(W)$$

A brick $B$ can be either simple or composite. A composite brick $CB$ is an architecture in its
own right, allowing for multiple levels of architectural abstraction. We omit the formal definition of
$CB$ for brevity; the definition is essentially the same as that for architecture $A$ above. Each simple
brick $SB$ is a tuple comprising the brick’s internal state $S$, its interface $I$, set of operations $O$, the
map $M$ that relates the operations and the interfaces through which they are exported, and the
probability distribution $\theta$ over the system’s topics $T$.

$$B = SB \cup CB$$
$$SB = \{b_i \mid i \in \mathbb{N}\}$$
$$b = (S, I, O, M, \theta_b)$$

A brick’s state $S$ is defined as a set of variables $var$, where each variable is a tuple comprising
a name $n$ (which must be one of the words in $W$), a type $t$, and a value $val$. 
\[ S = \{ v_i \mid i \in \mathbb{N} \} \]
\[ v = (n, t, val) \]
\[ n \subseteq W \]

A brick’s interface consists of a set of interface elements \( ie \), each of which is a tuple comprising a name \( ni \) (which must be one of the words in \( W \)), a possibly empty set of parameters \( P \), and a possibly empty set of return variables \( RV \).

\[ \mathbb{I} = \{ ie_i \mid i \in \mathbb{N} \} \]
\[ ie = (ni, P, RV) \]
\[ ni \subseteq W \]
\[ P = \{ v_j \mid j \in \mathbb{N}_0 \} \]
\[ RV = \{ v_k \mid k \in \mathbb{N}_0 \} \]

A brick’s operation \( op \) is a tuple comprising a set \( VO \) of variables that comprise the operation’s state, an algorithm \( alg \) that realizes the operation, a probability distribution \( \theta_{op} \) over the operation’s topics (called “document-topic distribution” for short) and a function \( op\_type \). \( T_{op} \) are the set of topics over which \( \theta_{op} \) is distributed, i.e., \( T_{op} \) represents the operation’s concerns. \( op\_type \) determines whether a topic in \( T_{op} \) is application-specific (pertaining to the system’s “business logic”) or application-independent (pertaining to the bricks’ interaction needs).

\[ \mathbb{O} = \{ op_i \mid i \in \mathbb{N} \} \]
\[ op = (VO, alg, \theta_{op}, op\_type) \]
\[ VO = \{ v_l \mid l \in \mathbb{N}_0 \} \]
\[ vo = \{ v_g \mid g \in \mathbb{N} \} \]
\[ \theta_{op} = Pd(T_{op}) \]
\[ T_{op} = \{ z_j \mid j \in \mathbb{N} \} \]
\[ op\_type : T_{op} \rightarrow SP \]
\[ SP = \{ spec, indep \} \]

The mapping relation \( M \) relates a brick’s operations with the interface elements through which they are accessed. The tuples in the relation are restricted such that every interface is paired with an operation in the tuple if their types match. Note that multiple operations can be part of different tuples comprising the same interface.

\[ M = \{ (ie_k, op_j) \mid \forall ie_k \in \mathbb{I} \} \]
\[ \exists op_j \in \mathbb{O} \mid \forall v_m \in ie_k.P \cup ie_k.RV \exists v_h \in op_j.s_{op} \mid v_h.t = v_m.t \}

The document-topic distribution \( \theta_b \) is a probability distribution over topics \( T \). \( \theta_b \) represents the extent to which the concerns represented by topics \( T \) are present within the brick \( b \).

\[ \theta_b = Pd(T) \]
A link \( l \) is a tuple comprising a source interface \( src \) and a destination interface \( dst \). Links are the channels over which components and connectors transfer data and control over their interfaces.

\[
L = \{ l_i \mid i \in \mathbb{N}_0 \} \\
l = (src, dst) \\
src, dst \in \mathbb{I}
\]

A component \( c \) is a brick whose interfaces are all application-specific and whose topics are primarily application-specific. Interfaces are considered application-specific if the words naming the interface are application-specific. In particular, each word of the system is classified as either application-independent or application-specific. A brick is considered primarily application-specific if, for each topic that occurs in the component with a probability above a threshold \( th_{zc} \), that topic is application-specific. \( th_{zc} \) is specified by an architect.

\[
C = \{ c_i \mid c_i \in B \land i \in \mathbb{N} \land c_i, \mathbb{I} = \mathbb{I} \} \\
I = \{ ia_j \mid j \in \mathbb{N} \land ia_j \in \mathbb{I} \land ia_j, n \subset \mathbb{AS} \} \\
AS = \{ w_j \mid j \in \mathbb{N} \land w_j \in \mathbb{W} \land w.type(w_j) = spec \} \\
w.type: \mathbb{W} \to \mathbb{SP} \\
0 \leq th_{zc} \leq 1 \\
z.type: \mathbb{T} \to \mathbb{SP}
\]

A connector \( n \) is a brick whose interfaces are all application-independent. Interfaces are considered application-independent if the words naming the interface are application-independent. A brick is considered primarily application-independent if, for each topic that occurs in the connector with a probability above a threshold \( th_{zn} \), that topic is application-independent. \( th_{zn} \) is specified by an architect. \( TP_n(n) \) is a relation indicating the types a connector \( n \) may be.

\[
N = \{ n_i \mid n_i \in B \land i \in \mathbb{N} \land n_i, \mathbb{I} = \mathbb{D} \} \\
D = \{ id_j \mid j \in \mathbb{N} \land id_j \in \mathbb{I} \land id_j, n \subset \mathbb{AD} \} \\
AD = \{ w_k \mid k \in \mathbb{N} \land w_k \in \mathbb{W} \land w.type(w_k) = indep \} \\
w.type: \mathbb{W} \to \mathbb{SP} \\
0 \leq th_{zn} \leq 1 \\
TP_n(n) = \{ ty_i \mid i \in \mathbb{N} \land n \in \mathbb{N} \land ty_i \in \{ \text{proc} \_\text{call}, \text{event}, \text{stream}, \text{distributor}, \text{data}_\text{access}, \text{adaptor}, \text{arbitrator} \} \}
\]

2 Architectural Smell Definitions

Extraneous Adjacent Connector. The *Extraneous Adjacent Connector* smell occurs when two connectors of different types are used to link a pair of components. Components \( c_1, c_2 \in C \) and connectors \( n_1, n_2 \in N \) are involved in an instance of an *Extraneous Adjacent Connector* smell if:
Ambiguous Interface. **Ambiguous Interface** are interfaces or meta-interfaces that offer only a single, general entry-point to a component or connector; contain a single parameter; and dispatch to different internal operations based on the content of the parameter. An interface \( i \in \mathbb{I} \) of \( b \in B \) is an **Ambiguous Interface** if:

\[
|b, \mathbb{I}, p| = 1 \land |\{op \mid M(b, \mathbb{I}, p_1, op)\}| > 1
\]

Scattered parasitic functionality is a concern-based architectural smell. It describes a system in which multiple components are responsible for realizing the same high-level concern while some of those components are also responsible for additional, orthogonal concerns. Such an orthogonal concern “infects” a component, akin to a parasite. Formally, a set of components SPF \( \in C \) suffer from this smell if:

\[
\exists z \in T \mid (\text{numCompsWithTopic}(z) > \text{th}_{tc}) \land \\
(\forall c \in \text{SPF})(P(z \mid c) > \text{th}_{spf})
\]

where \( 0 \leq \text{th}_{spf} \leq 1 \) specifies the acceptable degree of scattering per concern; \( \text{th}_{tc} \) captures that scattering of a topic is allowed to occur across a given number of components before they are considered to be affected by this smell; and \( \text{numCompsWithTopic} \) returns the number of components that has concern \( z \) with a proportion above \( \text{th}_{spf} \).

Connector Envy. Components with **Connector Envy** encompass extensive interaction-related functionality that should be delegated to a connector. A component \( c \in C \) suffers from **Connector Envy** in the following cases:

- **Connector Interface Implementation.** In this case, a component exposes an application-independent interface. Interface \( ia \) of component \( c \) exhibits this smell if: \( \exists op \in c.\emptyset \mid ia \in c.\mathbb{I} \land \{ia, op\} \in c.M \land \text{op.type}(op) = \text{indep} \)

- **Unacceptably High Connector Concern.** In this case, a single application-independent concern as represented by a topic is too high as specified through a threshold selected by an architect. A component \( c \in C \) exhibits this smell case if: \( \exists z \in T \mid z.\text{type}(z) = \text{indep} \land P(z \mid c) > \text{th}_{z_n} \)
• **Data Flow Interface Envy.** For this smell case, a component’s interface simply passes the parameters of the interface as the return value of some other interface. Component interfaces $i_1, i_2 \in c.I$ exhibit this smell if: $\forall p_i \in i_1.P, \forall rv_i \in i_2.RV \mid i_1 \neq i_2 \land par_i = rv_i$

**Component Envy.** Connectors with **Component Envy** encompass extensive application-specific functionality that should be performed by a component. A connector $n \in N$ suffers from **Component Envy** in the following cases:

• **Envious Interface Implementation.** In this case, a connector exposes an application-specific interface. Interface $ia$ of connector $n \in N$ exhibits this smell if: $\exists op \in n.\mathcal{O} \mid ia \in n.\mathcal{I} \land \{ia, op\} \in n.M \land op_type(op) = spec$

• **Unacceptably High Application-Specific Concern.** In this case, an application-oriented concern as represented by a topic is too high as specified through a threshold selected by an architect. A connector $n \in N$ exhibits this smell case if: $\exists z \in T \mid z.type(z) = spec \land P(z \mid n) > th_zc$

**Concern overload** indicates that a component implements an excessive number of concerns. Formally, a component $c \in C$ suffers from this smell iff

\[
\{ \{z_j \mid (j \in \mathbb{N}) \land (P(z_j \mid c) > th_zc)\} \} > th_{co}
\]

where $0 \leq th_{zc} \leq 1$ is the threshold indicating that a topic significantly represented in the component; and $th_{co} \in \mathbb{N}$ is a threshold indicating the maximum acceptable number of concerns per component.

**Link Overload** is a dependency-based smell that occurs when a component has interfaces involved in an excessive number of links (i.e., dependencies on other components), affecting the system’s separation of concerns and effective isolation of changes. A component may have an excessive number of incoming links, outgoing links, or both. Formally, a component $c_i$ suffers from outgoing link overload iff

\[
| \{vl \in L \mid l.src \in c_i.\mathcal{I}\} | > th_{lo}
\]

where $th_{lo}$ is a threshold indicating the maximum number of links for a component that is considered to be reasonable. Excessive incoming links are defined analogously.

**Unused Interface.** A brick’s interface is unused if that interface is not used by another brick. A brick $b_1 \in B$ contains an **Unused Interface** $ie_1 \in b_1.\mathcal{I}$ if: $\neg \exists b_2 \in B \mid ie_2 \in b_2.\mathcal{I} \land b_1 \neq b_2 \land ((ie_1, ie_2) \in L \lor (ie_2, ie_1) \in L) \lor (ie_2, ie_1) \in L)$

**Duplicate Component Functionality.** A component has duplicated functionality if it shares the same functionality as another component. A component $c \in C$ has a duplicate functionality if $\exists ie_1 \in c.\mathcal{I} \land \exists b \in B \mid ie_2 \in b.\mathcal{I} \land c \neq b \land \forall v_q \in ie_1.P \cup ie_1.RV, v_r \in ie_2.P \cup ie_2.RV \mid v_q = v_r \land \exists op_1 \in \mathcal{O}, op_2 \in b.\mathcal{O} \mid (ie_1, op_1) \in c.M \land (ie_2, op_2) \in b.M$

**Dependency cycle** indicates a set of components whose links form a circular chain, causing changes to one component to possibly affect all other components in the cycle. Formally, this smell occurs in a set of three or more components $c_k \in C$ iff

\[
\exists l \in L \mid (\forall x \mid (1 \leq x \leq k) \mid \\
(x < k) \implies (l.src \in c_x.\mathcal{I} \land l.dst \in c_{x+1}.\mathcal{I})) \land \\
(x = k) \implies (l.src \in c_k.\mathcal{I} \land l.dst \in c_1.\mathcal{I})
\]

**Unused Brick.** A brick is unused if its interfaces are all unused interfaces.
**Connector Dimension Overload.** Connector Dimension Overload occurs when a connector or set of linked connectors contains an excessive number of connector dimensions or sub-dimensions. For example, a set of connectors that are linked and perform authentication, authorization, encryption, streaming, data access and distribution is an instance of Connector Dimension Overload.

**Lego Syndrome.** A brick suffers from Lego Syndrome when it handles an extremely small amount of functionality or a minor concern that is encompassed by a system concern. This smell type represents bricks that are excessively small.

**Sloppy Delegation.** Sloppy Delegation occurs when a component delegates to another component a small amount of functionality that it could have performed itself. For example, a component that stores an aircraft’s current velocity, fuel level, and altitude and passes that data to another component that solely calculates that aircraft’s burn rate is an example of Sloppy Delegation.

**Brick Functionality Overload.** A brick that performs an excessive amount of functionality suffers from Brick Functionality Overload. Note that the functionality performed by a brick is not necessarily mapped one-to-one to that brick’s concerns.